ESTRUCTURAS DATOS

Trabajo Práctico N° 2

Listas Simples (continuación)

02

2024

Apellido y Nombre: Fecha:

EJEMPLOS

Ejemplo 1 – Variante de implementación: Modifique la implementación básica de listas simples de enteros (con punteros de *inicio* y *final*) de modo que se incluya un elemento para registrar la cantidad de nodos almacenados. Además desarrolle las operaciones *iniciar_lista*, *agregar_final*, *quitar_inicio* y *mostrar_lista* adaptadas a la implementación propuesta. Tenga en cuenta que la operación *agregar_final* sólo añadirá un nuevo nodo si éste contiene un valor mayor 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 los punteros de inicio y final 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
        final:pnodo
        contador:ENTERO
      FIN_REGISTRO
```

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

La operación *iniciar_lista* se realiza mediante un procedimiento que inicializa la lista. La inicialización crea una lista vacía asignando a *inicio, final* y *contador* los valores adecuados. En este caso, a los punteros *inicio* y *final* se les 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.final←NULO
   num.contador←0
FIN
```

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

La operación *agregar_final* se realiza mediante un procedimiento que añade un nuevo nodo al final de la lista siempre que éste contenga un valor mayor a los ya almacenados. Para ello, se consideran 2 casos: una lista vacía y una lista con elementos.

Si la lista se encuentra vacía entonces los punteros *inicio* y *final* deberán apuntar al nuevo nodo, incrementándose el *contador* para indicar que se ha agregado un dato. En cambio, si la lista contiene elementos entonces debe modificarse el último elemento (apuntado por *final*) de modo que el puntero *sig* de éste apunte al nuevo nodo, actualizándose el puntero *final* para hacer referencia al *nuevo* e incrementándose el contador.

Al agregar un nodo a la lista debe verificarse que el valor de éste sea mayor a los ya almacenados. Para ello, puede compararse el nuevo valor con el último de la lista ya que los anteriores a éste serán menores. Si el nuevo nodo no puede agregarse entonces será liberado.

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

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

La operación *quitar_inicio* se realiza mediante una función que retorna la dirección de un nodo extraído del inicio 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 y se modifica el contador. En particular, si la lista contiene sólo un elemento la operación de extracción generará una lista vacía, modificando los punteros de *inicio* y *final*.

El nodo extraído es completamente desconectado de la lista haciendo NULO su puntero siguiente (*sig*).

```
FUNCIÓN quitar_inicio(E/S num:tlista):pnodo
VARIABLES
   aux:pnodo
INICIO
   SI num.contador=0 ENTONCES
      aux←NULO
   SINO
     SI num.contador=1 ENTONCES
        aux←num.inicio
        num.inicio←NULO
        num.final←NULO
        num.contador←0
     SINO
        aux←num.inicio
        num.inicio←num.inicio->sig
        aux->sig←NULO
        num.contador←num.contador-1
     FIN_SI
   FIN_SI
   quitar_final←aux
FIN
```

```
pnodo quitar_inicio(tlista &num)
{ pnodo aux;
  if (num.contador==0)
   aux=NULL;
  else
   { if (num.contador==1)
     { aux=num.inicio;
       num.inicio=NULL;
       num.final=NULL;
       num.contador=0;
     }
     else
     { aux=num.inicio;
       num.inicio=num.inicio->sig;
       aux->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.

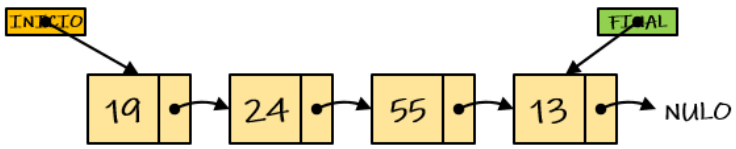
Obsérvese que el algoritmo utiliza como parámetro un dato de tipo *pnodo* y no *tlista*, esto permite que en cada llamado se trabaje con el mismo tipo de problema (un puntero pnodo). Así, el llamador original al procedimiento podría ser *mostrar_lista(**lista.inicio**)*.

```
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

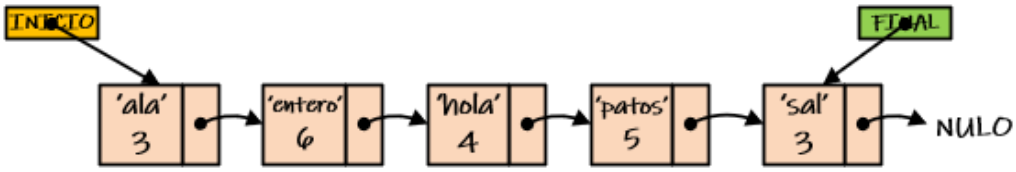
1) En base a la definición de *Lista Simple*, y considerando punteros al *inicio* y *final* de la lista, implemente sus operaciones fundamentales de teniendo en cuenta lo siguiente:
   - Una lista se compone de nodos, que almacenen datos y que poseen un indicador del próximo elemento de la lista.
   - 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 simple (con punteros de *inicio* y *final*) de caracteres, modifique la definición básica de modo que sea posible registrar la cantidad de mayúsculas, cantidad de minúsculas, cantidad de símbolos y cantidad de dígitos almacenados. Considere que la capacidad de la lista estará limitada a 5 caracteres de cada tipo (5 mayúsculas, 5 minúsculas, 5 símbolos y 5 dígitos). Adapte las operaciones *agregar_final*, *quitar_inicio* y *mostrar lista*.
3) Dada una lista de reales, con punteros de *inicio* y *final*, realice lo siguiente:
   a) Consigne la declaración de tipos y variables de la estructura.
   b) Diseñe un procedimiento/función que permita agregar un nuevo nodo a la lista por el *inicio* o *final* según un parámetro de opción. Considere que no podrán almacenarse valores repetidos. Los nodos que no puedan agregarse deben ser liberados.
   c) Diseñe un procedimiento/función que permita extraer un nodo de la lista por el *inicio* o *final* de acuerdo un parámetro de opción.
   d) Diseñe un procedimiento/función **recursiva** que permita buscar un valor en la lista. Considere 2 variantes para la implementación: una función lógica y una función de tipo puntero.
4) Dada la siguiente lista enlazada



a) Defina la estructura de datos correspondiente

b) Implemente operaciones que permitan
   - determinar si la lista no contiene valores repetidos
   - invertir el contenido de la lista
   - crear una copia de la lista

5) Dada una lista de cadenas de caracteres, con punteros de *inicio* y *final*, realice lo siguiente:



a) Consigne la declaración de tipos y variables de la estructura. Tenga en cuenta que debe registrarse la longitud de cada una de las cadenas almacenadas.

b) Implemente la operación *agregar_orden* considerando que no se admiten valores repetidos. Los nodos no insertados deben ser liberados. Tenga en cuenta que la búsqueda realizada para verificar valores repetidos debe aprovechar el criterio de orden de la lista, evitando recorridos completos innecesarios.

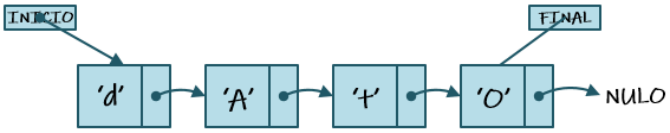
c) Implemente la operación *long_max* que muestre la palabra de mayor longitud almacenada.

6) Considerando listas simples (con punteros de *inicio* y *final*), analice los siguientes módulos y determine su propósito:

```
int enigma(pnodo inicio, pnodo fin)
{ if (inicio==NULL)
    return 0;
  else
  { if(inicio==fin)
      return 1;
    else
      return 1+enigma(inicio->sig,fin);
  }
}
void incognita(tlista lista,pnodo &b)
{
  b=lista.inicio;
  if (lista.inicio!=lista.final)
  { for(i=lista.inicio;i!=final;i=i->sig)
      if (b->dato < (i->sig)->dato)
          b=i->sig;
  }
}
```

```
bool secreto(tlista a,tlista b)
{ bool c=true;
  pnodo i,j;
  if (a.inicio!=NULL && b.inicio!=NULL)
  { i=a.inicio;
    j=b.inicio;
    while(i!=NULL && j!=NULL && c==true)
    { if (i->dato!=j->dato)
        c=false;
      i=i->sig;
      j=j->sig;
    }
    if (i!=j)
      c=false;
  }
  return c;
}
```

7) Dada la siguiente lista



a) defina las estructuras de datos que permitan representarla e

b) implemente las operaciones *contar_nodos* **recursiva,** *contar_caracteres* **recursiva** y *mostrar_lista* **recursiva** (desde el *inicio* o el *final* según parámetro de opción). Considere que *contar_caracteres* debe obtener la cantidad de mayúsculas, cantidad de minúsculas, cantidad de símbolos y cantidad de dígitos de la lista.

8) Dada la siguiente definición de listas

```
typedef struct tnodo *pnodo;
typedef struct tnodo{
                    char dato;
                    pnodo sig;
};
typedef pnodo tlista[2];
```

Implemente las operaciones

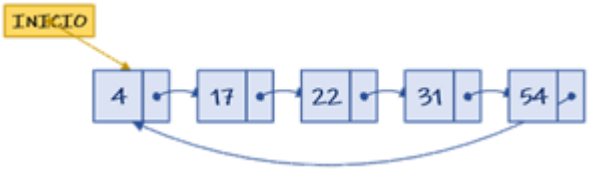
a) agregar en orden

b) quitar final

c) mostrar (recursivo)

d) ¿Cómo se modificarán la definición de datos y las operaciones anteriores si debe registrar la cantidad de elementos almacenados?

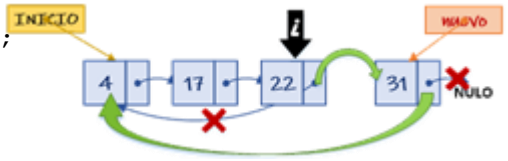
9) Dada la siguiente lista **CIRCULAR**:



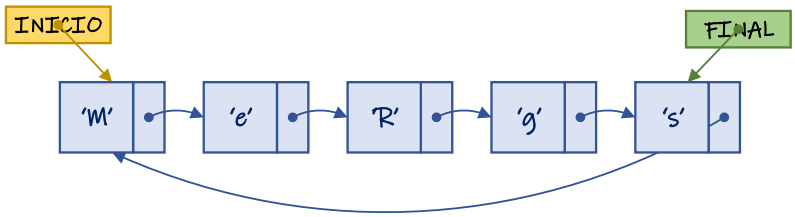
a) defina las estructuras de datos que permitan representarla.

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, mostrar_lista* y *buscar_valor*? 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;
   }
}
```



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



a) defina las estructuras de datos que permitan representarla, limitando su capacidad a 25 elementos.

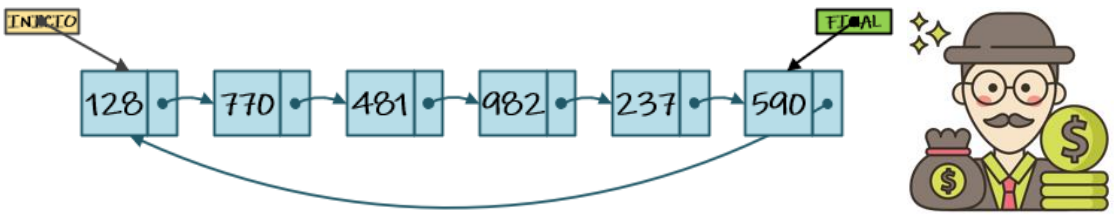
b) implemente las operaciones *iniciar_lista*, *agregar_final*, *quitar_nodo, ordenar y mostrar_lista*. Considere que la lista no puede contener valores repetidos.

11) Utilizando listas simples circulares, con punteros de *inicio* y *final*, diseñe un programa que permita simular el comportamiento de una pequeña ruleta. Para ello, considere que la ruleta se compone de *N* valores aleatorios distintos en el intervalo [100,999]. Al momento de jugar, el usuario deberá elegir 3 valores de la ruleta, hacerla girar (un valor aleatorio determinará cuantos nodos se recorren hasta finalizar los giros) y verificar si el valor obtenido corresponde a alguno de los 3 números jugados. Según sea el número acertado el usuario podrá ganar:

I. 5.000.000 de pesos, primer número jugado
II. 3.000.000 de pesos, segundo número jugado
III. 1.000.000 de pesos, tercer número jugado

Si el usuario pierde entonces debe mostrarse "NO TE RINDAS, INTENTA OTRA VEZ …".

Tenga en cuenta que la cantidad de números (*N*) de la ruleta será especificada por el usuario, y que la ruleta no puede contener valores repetidos.



12) El responsable del área de inventario de una cadena de electrodomésticos desea registrar información acerca de los productos comercializados por la empresa. Por cada producto se debe guardar: id de producto, nombre, descripción (marca, modelo, especificaciones, fabricante), precio unitario y stock. Considerando esto, se solicita:

a) Consigne la declaración de tipos y variables de la estructura que represente la situación planteada.

b) Diseñe los procedimientos/funciones que permitan listar los productos (nombre, modelo, precio y stock) que pertenezcan a un fabricante especificado por el usuario. Además, indique la cantidad de productos encontrados.

c) Diseñe los procedimientos/funciones que permitan mostrar el/los producto/s (nombre, marca, precio unitario y stock) con el mayor stock.

d) Diseñe los procedimientos/funciones que permitan modificar los datos de un producto específico. Considere que la operación debe solicitar la confirmación del usuario para registrar los cambios.