Estructuras de Datos Dinámicas

Resumiendo, son aquellas cuyo tamaño (longitud, número de elementos...) varía en tiempo de ejecución. Las más famosas son: Listas simplemente enlazadas (con y sin cabecera), Listas doblemente enlazadas (con y sin cabecera), Pilas, Colas, Árboles y Grafos.

A continuación se exponen algunos ejemplos de listas:

Tipo Lista Sin Cabecera

```
/*********************************************************/
/* Ejemplo de programa para manejar una:                 */
/*   LISTA SIMPLEMENTE ENLAZADA SIN CABECERA,            */
/*   como Estructura de Datos Dinámica (con punteros).   */
/* Incluye: Primitivas de la lista y programa de prueba. */
/*          Todo en el mismo fichero.                    */
/*********************************************************/
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>

/* Declaración de tipos */
struct element {
  long num;
  struct element *sig;
};

typedef struct element *lista;

/*****- PRIMITIVAS de una LISTA SIMPLEMENTE ENLAZADA SIN CABECERA -*****/

/*************************************************************/
/* Inicializa una lista simplemente enlazada y sin cabecera. */
void inicializar_lista(lista *list){
  *list=NULL;
}

/*************************************************************/
/* Devuelve TRUE si la lista está vacía.                     */
int lista_vacia(lista list){
  if (list) return 0;
  return 1;
}

/*************************************************************/
/* Devuelve la longitud de la lista list (núm. de elementos). */
long long_lista(lista list){
  long i=0;

  while (list) {
    list=list->sig;
    i++;
  }
  return i;
}
```

```
/*************************************************************/
/* Inserta el elemento e en la posición pos de la lista list.  */
/* Si pos<=1 inserta en la primera posición.                   */
/* Si pos>longitud_lista, inserta en la última posición.       */
/* Devuelve -1 si no hay memoria suficiente para la inserción. */
int insertar_pos (lista *list, struct element e, long pos){
  lista p, anterior, L=*list;
  long i=1;

  if ((p=(struct element *) malloc(sizeof(struct element))) == NULL)
     return -1;
  *p=e;

  if (pos<=1 || lista_vacia(*list)){
    /* Hay que insertar en la posición 1 */
    *list=p;
    p->sig=L;
    return 0;
  }

  while (L && i<pos){
    anterior=L;
    L=L->sig;
    i++;
  }

  /* Insertar elemento apuntado por p, entre anterior y L */
  anterior->sig=p;
  p->sig=L;
  return 0;
}


/*************************************************************/
/* Inserta el elemento e en la lista list por orden del campo num */
/* Devuelve -1 si no hay memoria suficiente para la inserción. */
int insertar_orden (lista *list, struct element e){
  lista p, anterior, L=*list;
  unsigned i=1;

  if ((p=(struct element *) malloc(sizeof(struct element))) == NULL)
     return -1;
  *p=e;

  if (e.num<=(*list)->num || lista_vacia(*list)){
    /* Si hay que insertar antes del primero o la lista está vacía */
    *list=p;
    p->sig=L;
    return 0;
  }

  while (L && e.num>L->num){
    anterior=L;
    L=L->sig;
    i++;
  }

  anterior->sig=p;
  p->sig=L;
  return 0;
}
```

```
/*************************************************************/
/* Borra el elemento en la posición pos de la lista list.    */
/* Si la pos=1 borra el primer elemento.                     */
/* Devuelve -1 si no existe la posición: pila vacía o pos>long */
int borrar_elemento(lista *list,long pos){
  lista aux, anterior, L=*list;
  long i=1;

  if (lista_vacia(*list) || pos<1)
     return -1;

  if (pos==1) { /* Tratamiento particular para borrar elemento 1 */
     (*list) = (*list)->sig;
     free(L);
     return 0;
  }

  while (L && i<pos){
    anterior=L;
    L=L->sig;
    i++;
  }

  if (L){ /* OJO: Aqui no vale decir: if (i==pos)... */
     /* Borrar elemento apuntado por L, teniendo un puntero al anterior */
     anterior->sig=L->sig;
     free(L);
     return 0;
  }
  return -1; /* La lista tiene menos de pos elementos */
}

/*************************************************************/
/* Libera la memoria ocupada por toda la lista.              */
void liberar_lista(lista *list){
  while (*list) {
    borrar_elemento (list,1);
  }
}

/*************************************************************/
/* Devuelve en e, el elemento que está en la posición pos    */
/* de la lista list.                                         */
/* Si no existe esa posición, devuelve -1. En otro caso 0.   */
int leer_element (lista list, long pos, struct element *e){
  long i=1;

  if (lista_vacia(list) || pos<1)
     return -1;

  while (list && i<pos) {
    list=list->sig;
    i++;
  }
  if (list){
     *e=*list;
     return 0;
  }
  return -1;
}
```

```
/*************************************************************/
/* Devuelve la posición de la primera ocurrencia del elemento  */
/* e en la lista list, a partir de la posición pos (inclusive).*/
/* Esta ocurrencia será considerada por el campo num.          */
/* Devuelve 0 si no ha sido encontrada.                        */
/* Con esta función, cambiando pos, podremos encontrar TODAS   */
/* las ocurrencias de un elemento en la lista.                 */
long posic_lista(lista list, struct element e, long pos){
  long i=1;

  while (list && i<pos) { /* Ir a la posición pos */
    list=list->sig;
    i++;
  }
  if (!list) return 0;

  while (list && e.num!=list->num) { /* Intentar encontrar el elemento */
    list=list->sig;
    i++;
  }
  if (!list) return 0;
  return i;
}

/*************************************************************/
/* Actualiza la posición pos de la lista list con el elemento e*/
/* Devuelve -1 si no existe esa posición.                      */
int actualiza_lista(lista list, struct element e, long pos){
  long i=1;

  if (pos<1) return -1;   /* Posición no válida   */
  while (list && i<pos) { /* Ir a la posición pos */
    list=list->sig;
    i++;
  }
  if (!list) return -1; /* Posición no existe */

  list->num=e.num; /* Actualización */
  return 0;
}
/* FIN de las PRIMITIVAS de la LISTA SIMPLEMENTE ENLAZADA SIN CABECERA */


/*************************************************************/
/* Imprime un elemento e por la Salida estándard.              */
void Imprimir_elemento (struct element e){
  printf("%li",e.num);
}

/*************************************************************/
/* Muestra todos los elementos de la lista list por su orden.  */
void mostrar_lista(lista list){
  struct element e;
  unsigned i=1,tama=long_lista(list);

  while (i<=tama) {
    printf("\nElemento %u: ",i);
    leer_element(list,i++,&e);
    Imprimir_elemento(e);
  }
}
```

```
/*************************************************************/
/* Muestra un menu de opciones.                              */
void menu(void){
  puts("\n\t\t***** MENU *****\n");
  puts("\tS. Destruir lista y SALIR.");
  puts("\t1. Inicializar lista.");
  puts("\t2. Insertar por orden.");
  puts("\t3. Insertar por posición.");
  puts("\t4. Mostrar lista.");
  puts("\t5. Borrar elemento.");
  puts("\t6. Longitud lista");
  puts("\t7. Ver si está vacía.");
  puts("\t8. Ver elemento n-ésimo.");
  puts("\t9. Posición de un elemento a partir de una dada.");
  puts("\t0. Modificar un elemento en una posición.");
  printf("\n\tOpcion: ");
}


/*************************************************************/
void main(){
  lista list;
  struct element e;
  char opcion='x';
  long pos;

  inicializar_lista(&list);

  while (opcion!='S' && opcion!='s'){
     menu();
     opcion=getche();

     switch (opcion) {
       case 's':
       case 'S':liberar_lista(&list);
                break;

       case '1':liberar_lista(&list);
                inicializar_lista(&list);
                break;

       case '2':printf("\nDame número: ");
                scanf("%li",&(e.num));
                if (insertar_orden(&list,e))
                   puts("\n\aERROR: No existe memoria suficiente.");
                break;

       case '3':printf("\nDame número: ");
                scanf("%li",&(e.num));
                printf("\nDame la posición: ");
                scanf("%li",&pos);
                if (insertar_pos(&list,e,pos))
                   puts("\n\aERROR: No existe memoria suficiente.");
                break;

       case '4':printf("\n----------------");
                mostrar_lista(list);
                puts("\n----------------");
                getch();
                break;
```

```
       case '5':printf("\nDame posición a borrar: ");
                scanf("%li",&pos);
                if (borrar_elemento(&list,pos))
                   puts("\n\aERROR: No existe esa posición.");
                break;

       case '6':printf("\n- Longitud: %u.",long_lista(list));
                break;


       case '7':if (lista_vacia(list))
                     puts("\n- Lista VACIA.");
                else puts("\n- Lista NO VACIA.");
                break;

       case '8':printf("\nDame posición a leer: ");
                scanf("%li",&pos);
                if (leer_element(list,pos,&e))
                   puts("\n\aERROR: No existe esa posición.");
                else
                   printf("\n- Elemento %li: %li.\n",pos,e.num);
                break;

       case '9':printf("\nDame número para buscar su posición: ");
                scanf("%li",&(e.num));
                printf("\nDame la posición a partir de la que buscar: ");
                scanf("%li",&pos);
                if (!(pos=posic_lista(list,e,pos)))
                   puts("\n\aERROR: No existe ese elemento a partir de esa
posición.");
                else
                   printf(
               "\n-La primera posición encontrada de ese elemento es: %i",
                   pos);
                break;

       case '0':printf("\nDame número para actualizar: ");
                scanf("%li",&(e.num));
                printf("\nDame la posición a modificar: ");
                scanf("%li",&pos);
                if (actualiza_lista(list,e,pos))
                   puts("\n\aERROR: No existe esa posición. No se ha
modificado.");
                break;
     }
  }

  puts("\n\t***** FIN *****");
}
```

Tipo Lista Con Cabecera

Ahora, implementamos una lista CON cabecera y además utilizando las ventajas de la compilación separada, es decir, separamos la implementación de las primitivas de la lista de la implementación del programa que utiliza la lista. Así, tenemos 3 ficheros:

`LSTCCLIB.H`	Contiene los tipos de datos y las cabeceras de las primitivas del tipo lista con cabecera. Es utilizado para incluirlo con #include en los siguientes dos ficheros.
`LSTCCLIB.C`	Contiene la implementación de las primitivas de la lista.
`PRUCC.C`	Programa de prueba de la lista con cabecera. Contiene la función `main()`.

Tras hacer estos tres ficheros, se debe crear un proyecto (menú Project del Borland C++) y añadir a este fichero (opción Add) los dos ficheros con extensión `.C`. Al compilar, compilará sólo el fichero o los ficheros que no están compilados y al generar el ejecutable lo generará con el nombre que le demos al proyecto, no con el nombre del fichero que contenga la función `main()`.

Al principio de cada fichero se ha incluido un comentario con su contenido y el nombre del fichero, para facilitar su identificación.

```
/*********************************************************/
/* Fichero: LSTCCLIB.H                                   */
/* Tipos de datos y cabeceras de las PRIMITIVAS para una   */
/*   LISTA SIMPLEMENTE ENLAZADA CON CABECERA.            */
/*********************************************************/
/* Declaración del tipo base de la lista */
struct element {
  long num;
};

/*****- PRIMITIVAS de una LISTA SIMPLEMENTE ENLAZADA CON CABECERA -*****/

/* Tipos de la lista: */
struct celda{ /* Tipo de cada celda de la lista */
  struct element elem;
  struct celda   *sig;
};

typedef struct celda *lista; /* Tipo lista: Puntero a celta */

int  Inicializar_lista(lista *list);
int  Lista_vacia(lista list);
int  Insertar_pos (lista ant, struct element e, long pos);
int  Insertar_orden (lista ant, struct element e);
int  Borrar_elemento(lista ant, long pos);
long Long_lista(lista list);
void Vaciar_lista(lista list);
void Destruir_lista(lista *list);
int  Leer_element (lista list, long pos, struct element *e);
long Posic_lista(lista list, struct element e, long pos);
int  Actualiza_lista(lista list, struct element e, long pos);
```

```
/*************************************************************/
/* Fichero: LSTCCLIB.C                                       */
/* PRIMITIVAS para el manejo y control de una:               */
/*   LISTA SIMPLEMENTE ENLAZADA CON CABECERA.                */
/* La declaración del tipo elemento (struct element) y       */
/* otros tipos están en el fichero lstcclib.h                */

/*************************************************************/
#include<stdlib.h>
#include"lstcclib.h"

/*************************************************************/
/* Inicializa una lista simplemente enlazada y CON cabecera.   */
/* Devuelve -1 en caso de ERROR.                               */
int Inicializar_lista(lista *list){
  if (((*list)=(struct celda *) malloc(sizeof(struct celda))) == NULL)
     return -1;
  (*list)->sig=NULL;
  return 0;
}

/*************************************************************/
/* Devuelve TRUE si la lista está vacía.                       */
int Lista_vacia(lista list){
  if (list->sig) return 0;
  return 1;
}

/*************************************************************/
/* Inserta el elemento e en la posición pos de la lista ant.  */
/* Si pos<=1 inserta en la primera posición.                   */
/* Si pos>longitud_lista, inserta en la última posición.       */
/* Devuelve -1 si no hay memoria suficiente para la inserción. */
int Insertar_pos (lista ant, struct element e, long pos){
  lista p, L=ant->sig;
  long i=1;

  if ((p=(struct celda *) malloc(sizeof(struct celda))) == NULL)
     return -1;
  p->elem=e;

  while (L && i<pos){ /* Hallar posición en la que insertar */
    ant=L;
    L=L->sig;
    i++;
  }

  ant->sig=p; /* Insertar elemento apuntado por p, entre anterior y L */
  p->sig=L;
  return 0;
}
```

```
/*************************************************************/
/* Inserta el elemento e en la lista ant ordenadamente por   */
/* el campo elem.num                                         */
/* Devuelve -1 si no hay memoria suficiente para la inserción. */
int Insertar_orden (lista ant, struct element e){
  lista p, L=ant->sig;
  unsigned i=1;

  if ((p=(struct celda *) malloc(sizeof(struct celda))) == NULL)
     return -1;
  p->elem=e;

  while (L && e.num>L->elem.num){ /* Hallar posición en la que insertar */
    ant=L;
    L=L->sig;
    i++;
  }


  ant->sig=p; /* Insertar elemento apuntado por p, entre anterior y L */
  p->sig=L;
  return 0;
}

/*************************************************************/
/* Borra el elemento en la posición pos de la lista ant.     */
/* Si la pos=1 borra el primer elemento.                     */
/* Devuelve -1 si no existe la posición:                     */
/*      pila vacía, pos<1 o pos>long                         */
int Borrar_elemento(lista ant, long pos){
  lista aux, L=ant->sig;
  long i=1;

  if (Lista_vacia(ant) || pos<1)
     return -1; /* Posición NO válida */

  while (L && i<pos){ /* Situarse en la posición a borrar */
    ant=L;
    L=L->sig;
    i++;
  }

  if (L){
     /* Borrar elemento apuntado por L, teniendo un puntero al anterior */
     ant->sig=L->sig;
     free(L);
     return 0;
  }
  return -1; /* La lista tiene menos de pos elementos */
}
```

```
/*****************************************************************/
/* Devuelve la longitud de la lista list (núm. de elementos).  */
long Long_lista(lista list){
  long i=0;

  list=list->sig;
  while (list) {
    list=list->sig;
    i++;
  }
  return i;
}

/*****************************************************************/
/* Vacía la lista totalmente, dejándola inicializada.          */
/* Tras esta operación NO es necesario inicializarla.          */
void Vaciar_lista(lista list){
  while (Borrar_elemento (list,1) != -1);
}

/*****************************************************************/
/* Destruye la lista totalmente, liberando toda la memoria.    */
/* Tras esto ES necesario Inicializar la lista para reusarla.  */
void Destruir_lista(lista *list){
  Vaciar_lista(*list);
  free(*list); /* Liberar la cabecera */
  *list=NULL;
}

/*****************************************************************/
/* Devuelve en e, el elemento que está en la posición pos      */
/* de la lista list.                                           */
/* Si no existe esa posición, devuelve -1. En otro caso 0.     */
int Leer_element (lista list, long pos, struct element *e){
  long i=1;

  if (Lista_vacia(list) || pos<1)
     return -1;

  list=list->sig; /* Nos saltamos la cabecera */
  while (list && i<pos) { /* Localizar la posición pos */
    list=list->sig;
    i++;
  }


  if (list){
     *e=list->elem;
     return 0;
  }
  return -1;
}
```

```
/*************************************************************/
/* Devuelve la posición de la primera ocurrencia del elemento  */
/* e en la lista list, a partir de la posición pos (inclusive).*/
/* Esta ocurrencia será considerada por el campo elem.num      */
/* Devuelve 0 si no ha sido encontrada.                        */
/* Con esta función, cambiando pos, podremos encontrar TODAS   */
/* las ocurrencias de un elemento en la lista.                 */
long Posic_lista(lista list, struct element e, long pos){
  long i=1;

  list=list->sig; /* Saltar cabecera */
  while (list && i<pos) { /* Ir a la posición pos */
    list=list->sig;
    i++;
  }
  if (!list) return 0; /* No existe posición pos, luego... */

  while (list && e.num!=list->elem.num) { /* Intentar encontrar el
elemento */
    list=list->sig;
    i++;
  }
  if (!list) return 0; /* No encontrado */
  return i;            /* Encontrado en la posición i */
}

/*************************************************************/
/* Actualiza la posición pos de la lista list con el elemento e*/
/* Devuelve -1 si no existe esa posición.                      */
int Actualiza_lista(lista list, struct element e, long pos){
  long i=1;

  if (Lista_vacia(list) || pos<1) return -1;   /* Posición no válida   */

  list=list->sig; /* Saltar cabecera */
  while (list && i<pos) { /* Ir a la posición pos */
    list=list->sig;
    i++;
  }
  if (!list) return -1; /* Posición no existe */

  list->elem=e; /* Actualización */
  return 0;
}

/* FIN de las PRIMITIVAS de la LISTA SIMPLEMENTE ENLAZADA CON CABECERA */
```

```
/*************************************************************/
/* Fichero: PRUCC.C                                          */
/* Ejemplo de programa para manejar una:                     */
/*   LISTA SIMPLEMENTE ENLAZADA CON CABECERA.                */
/* Utiliza la librería lstcclib.h                            */
/*************************************************************/
#include<stdio.h>
#include<conio.h>
#include"lstcclib.h"

/*****************************************************************/
/* Imprime un elemento e por la Salida estándar.                 */
void Imprimir_elemento (struct element e){
  printf("%li",e.num);
}

/*****************************************************************/
/* Muestra todos los elementos de la lista list por su orden.  */
void mostrar_lista(lista list){
  struct element e;
  unsigned i=1,tama=Long_lista(list);

  while (i<=tama) {
    printf("\nElemento %u: ",i);
    Leer_element(list,i++,&e);
    Imprimir_elemento(e);
  }
}

/*****************************************************************/
/* Muestra un menu de opciones.                                  */
void menu(void){
  puts("\n\t\t***** MENU *****\n");
  puts("\tS. Destruir lista y SALIR.");
  puts("\t1. Inicializar lista.");
  puts("\t2. Insertar por orden.");
  puts("\t3. Insertar por posición.");
  puts("\t4. Mostrar lista.");
  puts("\t5. Borrar elemento.");
  puts("\t6. Longitud lista");
  puts("\t7. Ver si está vacía.");
  puts("\t8. Ver elemento n-ésimo.");
  puts("\t9. Posición de un elemento a partir de una dada.");
  puts("\t0. Modificar un elemento en una posición.");
  printf("\n\tOpcion: ");
}


/*****************************************************************/
void main(){
  lista list;
  struct element e;
  char opcion='x';
  long pos;

  Inicializar_lista(&list);
```

```
while (opcion!='S' && opcion!='s'){
   menu();
   opcion=getche();

   switch (opcion) {
     case 's':
     case 'S':Destruir_lista(&list);
          break;

     case '1':Vaciar_lista(list);
          break;

     case '2':printf("\nDame número: ");
          scanf("%li",&(e.num));
          if (Insertar_orden(list,e))
             puts("\n\aERROR: No existe memoria suficiente.");
          break;


     case '3':printf("\nDame número: ");
          scanf("%li",&(e.num));
          printf("\nDame la posición: ");
          scanf("%li",&pos);
          if (Insertar_pos(list,e,pos))
             puts("\n\aERROR: No existe memoria suficiente.");
          break;

     case '4':printf("\n----------------");
          mostrar_lista(list);
          puts("\n----------------");
          getch();
          break;

     case '5':printf("\nDame posición a borrar: ");
          scanf("%li",&pos);
          if (Borrar_elemento(list,pos))
             puts("\n\aERROR: No existe esa posición.");
          break;

     case '6':printf("\n- Longitud: %u.",Long_lista(list));
          break;

     case '7':if (Lista_vacia(list))
                puts("\n- Lista VACIA.");
          else puts("\n- Lista NO VACIA.");
          break;

     case '8':printf("\nDame posición a leer: ");
          scanf("%li",&pos);
          if (Leer_element(list,pos,&e))
             puts("\n\aERROR: No existe esa posición.");
          else
             printf("\n- Elemento %li: %li.\n",pos,e.num);
          break;
```

```
       case '9':printf("\nDame número para buscar su posición: ");
            scanf("%li",&(e.num));
            printf("\nDame la posición a partir de la que buscar: ");
            scanf("%li",&pos);
            if (!(pos=Posic_lista(list,e,pos)))
               puts(
              "\n\aERROR: No existe ese elemento a partir de esa posición.");
            else
               printf(
             "\n-La primera posición encontrada de ese elemento es: %i",
                pos);
            break;

       case '0':printf("\nDame número para actualizar: ");
            scanf("%li",&(e.num));
            printf("\nDame la posición a modificar: ");
            scanf("%li",&pos);
            if (Actualiza_lista(list,e,pos))
               puts("\n\aERROR: No existe esa posición. No se ha
modificado.");
            break;
     }
  }

  puts("\n\t***** FIN *****");
}
```