```
1  ------------------------------------------------------------------------
2  /*  Autores:
3   *  Carlos Alejandro Sivira Munoz   15-11377
4   *  Cesar Alfonso Rosario Escobar   15-11295
5   *
6   ------------------------------------------------------------------------
7   * Archivo: hasht.h
8   *
9   * Archivo de cabecera para la estructura hasht
10  *
11  */
12 #include "singly.h"
13
14 #ifndef __HASHT__
15 #define __HASHT__
16
17 #define HASH_SIZE 419
18
19 /* Tipo: hasht
20  * --------------
21  * Tabla de hash, se implimenta como un arreglo de listas singly de
22  * tamano igual a un primo no muy grande
23  */
24 typedef singly * hasht[HASH_SIZE];
25
26 /* hashtInit
27  * --------------
28  * Da un valor inicial a cada entra de la tabla de hash
29  *
30  *  h_: tabla de hash a inicializar
31  */
32 void hashtInit(hasht h_);
33
34 /* hashtDestroy
35  * --------------
36  * Libera la memoria de cada lista lista asociada a la tabla
37  *
38  *  h_: tabla de hash a destruir
39  *
40  */
41 void hashtDestroy(hasht h_) ;
42
43 /* hashtInsert
44  * --------------
45  * Inserta un elemento en la tabla
46  *
47  *  h_: tabla de hash que recibe el elemento
48  *  number: elemento a insertar
49  */
50 int hashtInsert(hasht h_, int number);
51
52 #endif
53 ------------------------------------------------------------------------
54 /*
55  * Archivo: hasht.h
56  *
57  * Archivo: Archivo fuente para la estructura de datos hasht
58  *
59  */
60 #include <stdio.h>
```

```c
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <dirent.h>
#include <string.h>
#include <unistd.h>
#include <string.h>
#include "hasht.h"

#define TRUE 1
#define FALSE 0

/*hashtInit
 * --------------
 * Da un valor inicial a cada entra de la tabla de hash
 *
 *  h_: tabla de hash a inicializar
 */
void hashtInit(hasht h_){
  int i_;

  for(i_=0; i_<HASH_SIZE; i_++)
    h_[i_] = (singly *) NULL;
}

/*hashDestroy
 * --------------
 * Libera la memoria de cada lista lista asociada a la tabla
 *
 *  h_: tabla de hash a destruir
 */
void hashtDestroy(hasht h_) {
  int i_;

  for(i_=0; i_<HASH_SIZE ; i_++) {
    if (h_[i_] != NULL)
      singlyDestroy(h_[i_]);
  }
}

/*hashtInsert
 * --------------
 * Inserta un elemento en la tabla
 *
 *  h_: tabla de hash que recibe el elemento
 *  number: elemento a insertar
 */
int hashtInsert(hasht h_, int number){
  int index;
  snode *dummie;

  index = number % HASH_SIZE;

  if (h_[index] != NULL ) {


    if ( singlySearch(h_[index], number)  ){
      return FALSE;
```

```
121     }
122
123   }else {
124     h_[index] = (singly *) malloc( sizeof(singly) );
125
126     singlyInit( h_[index] );
127   }
128
129
130   dummie = (snode *) malloc( sizeof(snode) );
131   snodeInit(dummie, number);
132
133   singlyInsert( h_[index], dummie);
134
135   return TRUE;
136 }
137 ----------------------------------------------------------------------
138 /*
139  * Archivo: list.h
140  *
141  * Descripcion: Implementacion de lista doblemente enlazada que contiene
142  *        palabras con recuencias asociadas.
143  *
144  */
145 #ifndef __LIST__
146 #define __LIST__
147 /*Tipo: node
148  *------------
149  *  Tipo para manejar a los elementos de una lista de frecuencia de palabras
150  *
151  *  word: palabra asociada con el nodo
152  *  frequency: frecuencia de la palabra en la lista
153  *  next: apuntador al elemento siguiente
154  *  prev: apuntador al elemento anterior
155  */
156 typedef struct list_element{
157   /*se define el struct "list_element" para poder referenciarlo dentro de si
158   mismo*/
159   char *word;
160   int frequency;
161   struct list_element *next;
162   struct list_element *prev;
163 } node;
164 /*Funcion: nodeInit
165  *------------
166  *  Inicializa un elemento de lista
167  *
168  *  e: apuntador al elemento a inicializar
169  *  c: cadena de caracteres a insertar
170  */
171 void nodeInit(node *e, char *c, int f);
172 /*Funcion: nodeSwap
173  *------------
174  *  Cambia los valores de los atributos "word" y "frequency" entre dos nodos
175  *
176  *  u,v: apuntadores a los nodos a intercambiar.
177  */
178 void nodeSwap(node *u, node *v);
179 /*Tipo: list
180  *------------
```

```
 *  Tipo definido para manejar la estructura lista
 *
 *  head: cabecera de la lista
 *  tail: cola de la lista
 *  size: tamano de la lista
 */
typedef struct {
  node *head, *tail;
  int size;
} list;
/*Funcion: listInit
 *------------
 *  Inicializa una lista
 *
 *  l: apuntador a la lista a inicializar
 */
void listInit(list *l);
/*Funcion: listSearch
 *------------
 *  Busca un elemento en una lista dada
 *
 *  l: apuntador a la lista en donde se realizara la busqueda
 *  e: apuntador al elemento a buscar
 *
 *  retorna: Si encuentra el elemento devuelve un apuntador a el, de lo
 *  contrario retorna NULL.
 */
node* listSearch(list *l, node *e);
/*Funcion: listInsert
 *------------
 *  Inserta, o aumenta la frecuencia de un elemento en la lista
 *
 *  l: apuntador a la lista en donde se insertara el elemento
 *  e: apuntador al elemento a insertar
 *
 *  retorna: Un entero que representa si el elemento fue insertado, o se
 *  aumento su
 *  frecuencia
 */
int listInsert(list *l, node *e);
/*Funcion: listSort
 * ------------
 *  Ordena los elementos de la lista por frecuencia
 *
 *  l: apuntador a la lista a ordenar
 */
void listSort(list *l);
/*  Funcion: listMerge
 *  --------------
 *  Convina de forma ordenada dos listas.
 *
 *  list_a: Lista de palabras con orden alfanumerico y por frecuencia.
 *  list_b: Lista de todas las palabras con orden alfanumerico y por frecuencia.
 *
 *  return: void.
 */
int listMerge(list *list_a, list *list_b);
/*Funcion: listPrint
 *------------
 *  Imprime en consola el contenido de la lista
```

```
 *
 *  l: lista a imprimir
 */
void listPrint(list *l);
#endif
--------------------------------------------------------------------------
/*
 * Archivo: list.c
 *
 * Descripcion: Implementacion de lista doblemente enlazada que contiene
 *        palabras con recuencias asociadas.
 *
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "list.h"
/*Funcion: nodeInit
 * ------------
 *  Inicializa un elemento de lista, asignando los valores iniciales
 *  de sus atributos.
 *
 *  e: apuntador al elemento a inicializar
 *  c: cadena de caracteres insertar
 */
void nodeInit(node *e, char* c, int f) {
  e->next = NULL;
  e->prev = NULL;
  e->frequency = f;
  e->word = c;
}
/*Funcion: nodeSwap
 * ------------
 *  Cambia los valores de los atributos "word" y "frequency" entre
 *  dos nodos
 *
 *  u,v: apuntadores a los nodos a intercambiar
 */
void nodeSwap(node *u, node *v) {
  char *temp_word;
  int temp_frequency ;

  temp_word = u->word;
  u->word = v->word;
  v->word = temp_word;

  temp_frequency = u->frequency;
  u->frequency = v->frequency;
  v->frequency = temp_frequency;
}
/*Funcion: listInit
 * ------------
 *  Inicializa una lista.
 *
 *  l: apuntador a la lista a inicializar
 */
void listInit(list *l){
  l->head = l->tail = NULL;
  l->size = 0;
}
```

```
/*Funcion: listSearch
 * ------------
 *  Busca un elemento en una lista dada. Para esto, compara el atributo
 *  "word" del elemento dado, con el atributo "word" de cada nodo en la
 *  lista.
 *
 *  l: apuntador a la lista en donde se realizara la busqueda
 *  e: apuntador al elemento a buscar
 *
 *  retorna: Si encuentra el elemento devuelve un apuntador a el, de lo
 *  contrario retorna NULL
 */
node* listSearch(list *l, node *e) {
  /*dummie actua como un iterador sobre los elementos de la lista*/
  node *dummie = l->head;
  while(dummie != NULL) {
    if(strcmp(dummie->word, e->word) == 0){
      return dummie;
    }
    dummie = dummie->next;
  }
  return NULL;
}
/*Funcion: listInsert
 * ------------
 *  Inserta un elemento en una lista. La funcion se asegura de ajustar cada
 *  atributo de los elementos a cambiar, asi como atributos de la lista
 *  (de ser necesario)
 *
 *  l: apuntador a la lista en donde se insertara el elemento
 *  e: apuntador al elemento a insertar
 *
 *  retorna: Un entero que representa si el elemento fue insertado, o se
 *  aumento su frecuencia
 */
int listInsert(list *l, node *e) {
  if (l->size == 0) {
    l->head = e;
    l->tail = e;
    e->frequency++;
  }else {
    /*Incremente la frecuencia en funcion de si el elemento esta o no en la
    lista*/
    node *contains = listSearch(l, e);
    if (contains != NULL){
      if(e->frequency == 0) e->frequency++;
      contains->frequency += e->frequency;
      return -1;
    }else {
      l->tail->next = e;
      e->prev = l->tail;
      l->tail = e;
      if(e->frequency == 0) e->frequency++;
    }
  }
  l->size++;
  return 1;
}
/*Funcion: listSort
 * ------------
```

```
 *  Ordena los elementos de la lista en forma decreciente por frecuencia,
 *  y luego, los elementos con la misma frecuencia se ordenan
 *  alfanumericamente. Todo esto  usando una modificacion del
 *  algoritmo "insertion Sort"
 *
 *  l: apuntador a la lista a ordenar
 */
void listSort(list *l) {
  if (l->size > 1) {
    /*Se define a i como nodo, para poder usar una copia de*/
    /* sus atributo mientras que j se define como apuntador para*/
    /* realizar modificaciones los atributos del nodo apuntado*/
    node i = *(l->head)->next;
    node *j ;
    /*En el caso en el que 2 elementos tengan la misma*/
    /*frecuencia, se comparan sus atributos "word" para ver*/
    /* ordenarlos alfanumericamente.*/
    while (&i != NULL) {
      j = i.prev;
      while(j!=NULL && (i.frequency >= j->frequency) ) {
        if (i.frequency > j->frequency)
          nodeSwap(j->next,j);
        else
          if ( strcmp(i.word,j->word) < 0 )
            nodeSwap(j->next,j);
          else
            break;
              j=j->prev;
      }

      if (i.next == NULL)
        break;
      i=*(i.next);
    }
  }
}
/*  Funcion: listMerge
 *  --------------
 *  Combina de forma ordenada dos listas.
 *
 *  list_a: Lista de palabras con orden alfanumerico y por frecuencia.
 *  list_b: Lista de todas las palabras con orden alfanumerico y por frecuencia.
 *
 *  return: void.
 */
int listMerge(list *list_a, list *list_b){
    node *node_b, *new_node;
    node_b = list_b->head;
  /*Si la lista_a esta vacia, se copia la lista_b en lista_a*/
  if(list_a->size == 0){
    list_a->head = list_b->head;
    list_a->tail = list_b->tail;
    list_a->size = list_b->size;
    return;
  /*Si la lista_b esta vacia, no se realiza mezcla alguna*/
  }else if(list_b->size == 0){
    return;
  /*Se insertan los elementos de la lista b en la lista a*/
  }else{
    while(node_b != NULL){
```

```
      new_node = malloc(sizeof(node));
      if(!new_node) {
        perror("MALLOC");
        return;
      }
      nodeInit(new_node, node_b->word, node_b->frequency);
      listInsert(list_a, new_node);
      node_b = node_b->next;
    }
  }
}
/*Funcion: listPrint
 * ------------
 *  Imprime en consola el contenido de la lista
 *
 *  l: lista a imprimir
 */
void listPrint(list *l_) {
  if (l_->size > 0) {
    node *dummie = l_->head;
    while (dummie != NULL ) {
      fprintf(stderr,"%s %d\n",dummie->word,dummie->frequency);
      if (l_->head == l_->tail)
        break ;
      dummie = dummie->next;
    }
  }
}
--------------------------------------------------------------------------------
/*
 * Archivo: myFind.h
 *
 * Descripcion: Archivo de cabecera para las funciones asociadas al recorrido
 * de directorio
 *
 */

#include "hasht.h"

#ifndef __MY_FIND__
#define __MY_FIND__

/* arrangeMod
 * --------------
 * Recibe dos enteros y calcula el modulo del maximo entre el minimo de ellos
 *
 *  a: entero a considerar
 *  b: entero a considerar
 *
 */
int arrangeMod(int a, int b);

/* extendWord
 * --------------
 *  Actualiza la memoria alojada para un arreglo de string
 *
 *    paths: arreglo de strings
 *    next_ceil: referencia al tamano actual
 *
 */
```

```
481 void extendWord(char*** paths, int next_ceil);
482
483 /* isTxt
484  * --------------
485  * Verifica que el sufijo de un string sea ".txt"
486  *
487  *  name: string cuyo sufijo se verifica
488  *
489  */
490 int isTxt(char *name) ;
491
492 /* traverseDir
493  * --------------
494  * Funcion que recorre directorios en busqueda de archivos que coincidan
495  * con un criterio de busqueda.
496  *
497  * Acumula estos archivos en un char**
498  *
499  *
500  *    dir: Apuntador a directorio que se recorre
501  *    dirname: nombre del directorio que se recorre
502  *    inodes: tabla de hash para inodos
503  *    paths: direccion de memoria del arreglo de strings
504  *    ind: Entero usado para indizar el proximo nombre de archivo a guardar
505  *
506  * Retorna el numero de archivos que cumplen los criterios mencionados
507  */
508 void traverseDir(DIR *dir, char *dirname, hasht *inodes, char*** paths, int ind) ;
509
510 /* myFind
511  * --------------
512  * Busca archivos con el sufijo .txt en una jerarquia de directorios,
513  * verifica que ninguno sea un hard link de otro y recupera la lista
514  * de pathnames
515
516  *
517  *  dirname: nombre de la raiz del arbol de directorios
518  *  paths: direccion del arreglo de strings
519  *
520  * Retorna el numero de archivos que cumplen un criterio
521  */
522 int myFind (char *dirname, char*** Paths) ;
523
524 #endif
525 ------------------------------------------------------------------------
526 /*
527  * Archivo: myFind.c
528  *
529  * Descripcion: Archivo fuente para las funciones asociadas al recorrido
530  * de directorios
531  *
532  */
533
534 #include <stdio.h>
535 #include <stdlib.h>
536 #include <sys/types.h>
537 #include <sys/stat.h>
538 #include <fcntl.h>
539 #include <dirent.h>
540 #include <string.h>
```

```c
#include <unistd.h>
#include <string.h>
#include "hasht.h"

#define TRUE 1
#define FALSE 0

#define MAX_PATHNAME 5000

#define STANDARD_SIZE 419
#define REG_SIZE 419

#define MIN(a,b) ((a < b)? a: b)
#define MAX(a,b) ((a > b)? a: b)

/* arrangeMod
 * --------------
 * Recibe dos enteros y calcula el modulo del maximo entre el minimo de ellos
 *
 *  a: entero a considerar
 *  b: entero a considerar
 */
int arrangeMod(int a , int b) {
  return MAX(a,b) % MIN(a,b);
}

/* extendWords
 * --------------
 * Actualiza la memoria alojada para un arreglo de string si se acerca a
 * un multiplo de su tamano alojado actual
 *
 *  paths: arreglo de strings
 *  next_ceil: multiplo de su tamano actual
 *
 */
void extendWords(char*** paths, int next_ceil) {
  if( ( *paths = (char**) realloc(*paths, sizeof(char*) *
          (next_ceil + REG_SIZE) ) ) == NULL)
    perror("realloc");
}

/* isTxt
 * --------------
 * Verifica que el sufijo de un string sea ".txt"
 *
 *  name: string cuyo sufijo se verifica
 */
int isTxt(char *name) {
  int n = strlen(name);

  if ( n>4 && (name[n-1] == 't') && (name[n-3] == 't') &&
      (name[n-2] == 'x') && (name[n-4] == '.') ) {
    return TRUE;
  }

  return FALSE;
}

/* traverseDir
 * --------------
```

```
 * Esta funcion recorre de forma recursiva un arbol de directorios dado un
 * directorio raiz, y anade a un arreglo de strings aquellos nombres de
 * archivo que tengan el sufijo ".txt".
 *
 * Para evitar repeticiones por hard links, emplea una tabla de hash
 * en la que guarda los inodos de cada nuevo archivo encontrado, asi
 * si se encuentra otro hard link de un archivo ya revisado, este se evita.
 *
 * Se utiliza un char*** para poder actualizar el tamano del arreglo de
 * strings en caso de que se alcance su cota base, y la nueva direccion se
 * preserve luego de la llamada a la funcion.
 *
 *
 *    dir: Apuntador a directorio que se recorre
 *    dirname: nombre del directorio que se recorre
 *    inodes: tabla de hash para inodos
 *    paths: direccion de memoria del arreglo de strings
 *    ind: Entero usado para indizar el proximo nombre de archivo a guardar
 *
 * Retorna el numero de archivos que cumplen los criterios mencionados.
 */
int traverseDir(DIR *dir, char *dirname, hasht inodes, char*** paths, int ind) {
  int len, term, help;
  char path[MAX_PATHNAME], pathname[MAX_PATHNAME];
  DIR *curr_dir;
  struct dirent *current_ent;
  struct stat current_st;


  help = 0;
  term = 0;
  strcpy(path, dirname);

  while( (current_ent = readdir(dir)) != NULL ) {

    if( strcmp(current_ent->d_name,".")!=0
        && strcmp(current_ent->d_name,"..")!=0)
    {
      strcpy(pathname, path);
      len = strlen( pathname ) ;

      if (pathname[len-1] != '/')
        strcat(pathname,"/");

      /* Extiendo el pathname para que incluya el nombre de la
       * entrada actual */
      strcat(pathname, current_ent->d_name);

      if(lstat(pathname, &current_st) == -1)
        perror("stat");


      /*Se ignoraran links simbolicos*/
      if ( S_ISREG(current_st.st_mode) &&
                isTxt(current_ent->d_name) &&
           hashtInsert(inodes,current_st.st_ino) )
      {

        if ((arrangeMod(STANDARD_SIZE, ind+term+1) == 0)
            && (ind + term + 1 != 1))
```

```
661         {
662           extendWords(paths, ind + term + 1);
663         }
664 
665         paths[0][ind+term] = (char*) malloc(sizeof(char)*
666                   strlen(pathname) + 1);
667 
668         if (paths[0][ind+term] == NULL)
669           perror("malloc");
670 
671         strcpy( paths[0][ind+term],pathname);
672         term++;
673       }
674       else if ( S_ISDIR(current_st.st_mode) &&
675           !S_ISLNK(current_st.st_mode) )
676       {
677           if( (curr_dir = opendir(pathname) ) == NULL)
678           perror("opendir ");
679 
680 
681         term += traverseDir(curr_dir, pathname,
682             inodes, paths, ind + term );
683 
684         closedir(curr_dir);
685       }
686 
687     }
688   }
689 
690   help += term;
691   return help;
692 }
693 
694 /* myFind
695  * --------------
696  * Busca archivos con el sufijo .txt en una jerarquia de directorios,
697  * verifica que ninguno sea un hard link de otro y recupera la lista
698  * de pathnames
699  *
700  *  dirname: nombre de la raiz del arbol de directorios
701  *  paths: direccion del arreglo de strings
702  *
703  * Retorna el numero de archivos que cumplen un criterio.
704  */
705 int myFind (char *dirname, char ***paths) {
706 
707   int n_paths;
708   DIR *dir;
709   struct stat d_stat;
710   hasht inodes;
711 
712   hashtInit(inodes);
713 
714   if ( stat(dirname, &d_stat) == -1 )
715     perror("stat");
716 
717   if( !S_ISDIR(d_stat.st_mode) ) {
718     fprintf(stderr,"Not a directory\n");
719     exit(-1);
720   }
```

```
  else {
    dir = opendir(dirname);
    n_paths = traverseDir(dir, dirname, inodes, paths, 0);
  }

  hashtDestroy(inodes);
  closedir(dir);

  return n_paths;
}
--------------------------------------------------------------------------------
/*
 * Archivo: singly.h
 *
 * Descripcion: Archivo de cabecera para la estructura singly.
 *
 */

#ifndef __SINGLY__
#define __SINGLY__

/* snode
 * --------------
 * Inicializa nodo de lista singly
 *
 *  n_: apuntador al nodo a inicializar
 *  number: inodo que acompana al nodo
 */
typedef struct singly_node {
  struct singly_node *next;
  int element;
} snode;

/* singly
 * --------------
 * Estructura singly
 *
 *  lista simplemente enlazada
 */
typedef struct {
  snode *head, *tail;
  int size;
} singly;

/* snodeInit
 * --------------
 * Inicializa nodo de lista singly
 *
 *  n_: apuntador al nodo a inicializar
 *  number: inodo que acompana al nodo
*/
void snodeInit(snode *n_, int number);

/* singlyInit
 * --------------
 * Inicializa una lista singly
 *
 *  l_: apuntador a una lista
 */
void singlyInit(singly *l_) ;
```

```
781
782 /* singlyDestroy
783  * --------------
784  * Libera la memoria de una lista singly
785  *
786  *  l_: lista a destruir
787  *
788  */
789 void singlyDestroy(singly *l_) ;
790
791 /* singlyInsert
792  * --------------
793  * Inserta un nodo en una lista singly
794  *
795  *  l_: apuntador a la lista en donde se insertara
796  *  n_: apuntador del nodo a insertar
797  *
798  */
799 void singlyInsert(singly *l_, snode *n_) ;
800
801 /* singlySearch
802  * --------------
803  * Revisa si un existe un nodo en la lista con un inodo asociado dado.
804  *
805  *  l_: apuntador a la lista
806  *  number: inodo a consultar
807  *
808  */
809 int singlySearch(singly *l_, int number) ;
810
811 #endif
812 ------------------------------------------------------------------------------
813 /*
814  * Archivo: singly.c
815  *
816  * Descripcion: Archivo fuente para las estructuras singly y snode
817  *
818  */
819 #include <stdio.h>
820 #include <stdlib.h>
821 #include <sys/types.h>
822 #include <sys/stat.h>
823 #include <sys/wait.h>
824 #include <fcntl.h>
825 #include <dirent.h>
826 #include <string.h>
827 #include <unistd.h>
828 #include <string.h>
829 #include "singly.h"
830
831 #define TRUE 1
832 #define FALSE 0
833
834 /* snodeInit
835  * --------------
836  * Inicializa nodo de lista singly
837  *
838  *  n_: apuntador al nodo a inicializar
839  *  number: inodo que acompana al nodo
840  */
```

```
841 void snodeInit(snode *n_, int number) {
842   n_->next = (snode *) NULL;
843   n_->element = number;
844 }
845
846 /* singlyInit
847  * --------------
848  * Inicializa una lista singly
849  *
850  *  l_: apuntador a una lista
851  */
852 void singlyInit(singly *l_) {
853   l_->head = l_->tail = (snode *) NULL;
854   l_->size = 0;
855 }
856
857 /* singlyDestroy
858  * --------------
859  * Libera la memoria de una lista singly
860  *
861  *  l_: lista a destruir
862  */
863 void singlyDestroy(singly *l_) {
864   snode *killed, *dummie;
865
866   if (l_->size >= 1 ) {
867     killed = l_->head;
868     dummie = killed->next;
869     while( dummie != (snode*) NULL ) {
870       free(killed);
871       killed = dummie;
872       dummie = dummie->next;
873     }
874     free(killed);
875   }
876
877   free(l_);
878 }
879
880 /* singlyInsert
881  * --------------
882  * Inserta un nodo en una lista singly
883  *
884  *  l_: apuntador a la lista en donde se insertara
885  *  n_: apuntador del nodo a insertar
886  */
887 void singlyInsert(singly *l_, snode *n_) {
888
889   if (l_->size == 0)  {
890     l_->head = l_->tail = n_;
891   }
892   else if (l_->size >= 1 )  {
893     (l_->tail)->next = n_;
894     l_->tail = n_;
895   }
896
897   l_->size++;
898 }
899
900 /* singlySearch
```

```
 * --------------
 * Revisa si un existe un nodo en la lista con un inodo asociado dado.
 *
 *  l_: apuntador a la lista
 *  number: inodo a consultar
 */
int singlySearch(singly *l_, int number) {
  int i_;
  snode *dummie;


  dummie = l_->head;
  for(i_=0; i_<l_->size; i_++) {

    if ( dummie->element == number)
      return 1;

    dummie = dummie->next;
  }

  return 0;
}
--------------------------------------------------------------------------------
/*
 * Archivo: main.c (frecpalhilo)
 *
 * Descripcion: Recibe un directorio, busca todos los archivos que terminan en
 *        .txt, lee su contendo y lo ordena el listas de frecuencia usando
 *        hilos.
 *
 */
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <pthread.h>
#include <semaphore.h>
#include <errno.h>
#include "list.h"

#define WORD_SIZE 20
#define MAX_THREADS 1000
#define STANDARD_SIZE 419
#define ARGV_DESP 0
#define SEM_COUNT 1
#define SEM_SHARED_WITH 0
/*----------------------------------ESTRUCTURAS-----------------------------*/
/*  Tipo: thread_vars_t
 *  -------------------
 *    Tipo para manejar los datos compartidos entre hilos.
 *
 *  files: Archivos quen contienen palabras a leer.
 *  main_list: Lista donde son mezclados las palabras.
 *  num_files: Numero total de archivos a leer.
 */
typedef struct thread_vars_struct{
    char **files;
  list *main_list;
  int num_files;
}thread_vars_t;
```

```
961 /*----------------------------------ESTRUCTURAS-----------------------------*/
962 /*-----------------------------VARIABLES GLOBALES--------------------------*/
963 int global_index;/*Contador de archivos leidos*/
964 sem_t sem_merge;/*Semaforo que controla el acceso a global_index*/
965 sem_t sem_index;/*Semaforo que controla el acceso a main_list*/
966 /*-----------------------------VARIABLES GLOBALES--------------------------*/
967 /*  freecpalList
968  *  ------------------
969  *    Cuenta el numero de ocurrencias de una palabra en un archivo. Para ello
970  *    lee cada palabra del archivo y la inserta en una "lista de frecuencias".
971  *
972  *    Recibe los archivos segun la disponibilidad del indice global.
973  *
974  *  arg: Structura que contiene la informacion que comparten los hilos
975  *
976  */
977 void *freecpalList(void *arg){
978   /*---------------------------------VARIABLES-----------------------------*/
979     thread_vars_t *vars;
980     FILE *fp;
981   int index;
982   char *current_word;
983   node *space;
984   list *my_list;
985   /*---------------------------------VARIABLES-----------------------------*/
986   /*Se salvan los datos compartidos como tipo thread_vars_t*/
987     vars = (thread_vars_t*)arg;
988   /*Se reserva el espacio de la lista del hilo*/
989   my_list = (list*)malloc(sizeof(list));
990   if (!my_list) {
991     perror("MALLOC");
992     pthread_exit((void*)-1);
993   }
994   /*Los hilos procesan archivos hasta agotar la existencia*/
995   do
996   {
997     listInit(my_list);
998     /*----------------------------REGION CRITICA ------------------------*/
999     sem_wait(&sem_index);
1000       index = (isFileAvailable(vars->num_files) == 1) ? global_index : -1;
1001    sem_post(&sem_index);
1002    /*----------------------------REGION CRITICA ------------------------*/
1003    if(index == -1){/*No hay mas archivos para leer*/
1004      free(my_list);
1005      pthread_exit((void*)0);
1006    }
1007
1008    if (!(fp = fopen(vars->files[index + ARGV_DESP],"r"))){
1009      fprintf(stderr, "%s",vars->files[index + ARGV_DESP]);
1010      perror("FOPEN");
1011      pthread_exit((void*)-3);
1012    }
1013
1014    current_word = (char*)malloc(WORD_SIZE*sizeof(char));
1015    if (!current_word) {
1016      perror("MALLOC");
1017      pthread_exit((void*)-1);
1018    }
1019    /*Lectura de palabras*/
1020    while(fscanf(fp,"%s",current_word) != EOF) {
```

```
      space = (node*)malloc(sizeof(node));
      if(!space) {
        perror("MALLOC");
        pthread_exit((void*)-1);
      }

      nodeInit(space, current_word, 0);
      if (listInsert(my_list, space) < 0) {
        free(space);
        free(current_word);
      }

      current_word = (char*)malloc(WORD_SIZE*sizeof(char));
      if(!current_word) {
        perror("MALLOC");
        pthread_exit((void*)-1);
      }
    }
    fclose(fp);
    /*-----------------------------REGION CRITICA ------------------------*/
    sem_wait(&sem_merge);
      listMerge(vars->main_list, my_list);
    sem_post(&sem_merge);
    /*-----------------------------REGION CRITICA ------------------------*/
    free(current_word);
  } while (index >= 0);
}
/*  isFileAvailable
 *  ---------------
 *    Verifica si hay archivos disponibles que un hilo pueda procesar.
 *
 *  n: Numero total de archivos a leer.
 *
 *  return: 1 si hay archivos disponibles. -1 en caso contrario.
 */
int isFileAvailable(int n){
  if(global_index < n - 1){
    global_index++;
    return 1;
  }else{
    return -1;
  }
}
/*  main
 *  --------------
 *    Cuenta el numero de ocurrencias de una palabra en un archivo. Para ello,
 *    mediante el uso de hilos, lee cada palabra de un archivo y las inserta
 *    en una "lista de frecuencias".
 *
 *    Muestra la frecuencia de las palabras en una lista principal que es el
 *    resultado de convinar las listas generadas por los hilos. Muestra esta
 *    lista en la salida estandar.
 *
 *  argc: Numero de argumentos.
 *  argv: <Numero de hilos> <Numero de archivos> {archivos con palabras}
 */
int main(int argc, char *argv[]){
  /*---------------------------------VARIABLES-----------------------------*/
  thread_vars_t *thread_vars;
  list *main_list;
```

```
1081      pthread_t *t_ids;
1082    char **paths;
1083      int i, n_thread, n_files, trash;
1084    /*---------------------------------VARIABLES----------------------------*/
1085    /*Se reserva el espacio a utilizar*/
1086    thread_vars = (thread_vars_t*)malloc(sizeof(thread_vars));
1087    main_list = (list*)malloc(sizeof(list));
1088    /*Se salvan los datos suministrados*/
1089    n_thread = atoi(argv[1]);
1090    /*Arreglo de identificadores de hilos*/
1091    t_ids = malloc(sizeof(pthread_t*)*n_thread);
1092    if (t_ids == NULL ) {
1093      perror("MALLOC");
1094      exit(-1);
1095    }
1096    global_index = -1; /*Se inicializa el contador*/
1097    /*Archivos a procesar*/
1098    paths = (char**) malloc(sizeof(char*) * STANDARD_SIZE);
1099    if (paths == NULL) {
1100      perror("MALLOC");
1101      exit(-1);
1102    }
1103    /*Cantidad de archivos a procesar*/
1104    n_files = myFind(argv[2], &paths);
1105    printf("numero de archivos encontrados: %d\n",n_files);
1106    printf("direccion de path %p\t tamano: %d\n",(void*) paths,
malloc_usable_size(paths));
1107
1108    /*Inicializacion de lista principal y semaforos*/
1109    listInit(main_list);
1110    sem_init(&sem_index, SEM_SHARED_WITH, SEM_COUNT);
1111    sem_init(&sem_merge, SEM_SHARED_WITH, SEM_COUNT);
1112    /*Inicializacion de datos compartidos entre hilos*/
1113    thread_vars->files = paths;
1114    thread_vars->main_list = main_list;
1115    thread_vars->num_files = n_files;
1116    /*Si el numero de hilos es superior al de archivos, se toma el numero de
1117    archivos como el numero de hilos a utilizar*/
1118    n_thread = (n_thread >= n_files ? n_files : n_thread);
1119
1120      /*Creacion de los hilos con la funcion freecpalList*/
1121      for (i = 0; i < n_thread; i++){
1122          if (pthread_create(&t_ids[i], NULL, *freecpalList, thread_vars) != 0){
1123              perror("PTHREAD");
1124        exit(-2);
1125          }
1126      }
1127
1128      /*Se realiza la espera de los hilos*/
1129      for (i = 0; i < n_thread; i++) {
1130          if (pthread_join(t_ids[i], NULL) != 0){
1131              perror("PTHREAD");
1132        exit(-2);
1133          }
1134      }
1135
1136    /*Muestra del contenido de la lista*/
1137    listSort(main_list);
1138    listPrint(main_list);
1139    /*Se libera la memoria*/
```

```
1140      sem_destroy(&sem_index);
1141    sem_destroy(&sem_merge);
1142    free(main_list);
1143    free(thread_vars);
1144    free(paths);
1145    free(t_ids);
1146    /*El hilo principal termina*/
1147      pthread_exit(NULL);
1148  }
1149  ------------------------------------------------------------------------
1150  /*
1151   * Archivo: main.c (frecpalproc)
1152   *
1153   * Descripcion: Archivo fuente para la rutina principal de la aplicacion
1154   * frecpalproc
1155   *
1156   */
1157  #include <stdio.h>
1158  #include <stdlib.h>
1159  #include <sys/types.h>
1160  #include <sys/stat.h>
1161  #include <sys/wait.h>
1162  #include <fcntl.h>
1163  #include <dirent.h>
1164  #include <string.h>
1165  #include <unistd.h>
1166  #include <string.h>
1167  #include <signal.h>
1168  #include <semaphore.h>
1169  #include <errno.h>
1170  #include "myFind.h"
1171  #include "list.h"
1172
1173  #define WRITE 1
1174  #define READ 0
1175
1176  #define WORD_SIZE 20
1177
1178  #define STANDARD_SIZE 419
1179
1180  #define SMP0 "/mutex"
1181  #define SMP1 "/sem_reader"
1182  #define SMP2 "/sem_writer"
1183  #define MODE S_IRWXU | S_IRGRP | S_IWGRP | S_IROTH
1184
1185  #define MIN(a,b) (a < b)? a: b;
1186
1187
1188  /* countFrequencies
1189   * --------------
1190   * Cuenta el numero de ocurrencias de una palabra en un archivo. Para ello
1191   * lee cada palabra del archivo y la inserta en una "lista de frecuencias"
1192   *
1193   * Recibe los archivos de un arreglo de strings, seleccionando solo un
1194   * segmento de este.
1195   *
1196   *
1197   *  my_list:  lista de frecuencias
1198   *  paths: arreglo con los nombres de los archivos a leer
1199   *  floor: cota inferior del arreglo de archivos a leer
```

```
 *  ceil: cota superior del arreglo de archivos a leer
 *  reference: se utiliza para desplazarse sobre el arreglo de string
 *
 */
void countFrequencies(list *my_list, char** paths, int floor, int ceil,
          int reference){
  int j_;
  FILE *fp;

  char* current_word;
  char* word_buffer;
  node *space;


  word_buffer = (char *) malloc( sizeof(char) * WORD_SIZE + 1);

  for(j_=floor*reference; j_ < floor*reference + ceil  ; j_++) {

    if ( !(fp = fopen(paths[j_],"r")) ){
      perror("fopen");

      exit(-3);
    }

    while( fscanf(fp,"%s", word_buffer) != EOF) {

      if ( ( space = (node*) malloc( sizeof(node) ) ) == NULL)
        perror("malloc");

      current_word = (char *) malloc( sizeof(char) *
          strlen(word_buffer) + 1 );
      if(current_word == NULL)
        perror("current_word ");

      strcpy(current_word, word_buffer);
      nodeInit(space, current_word, 1);

      /* En caso de que solo la frecuencia de un elemento
       * se aumente como ese nodo ya esta creado, se libera
       * la memoria que se almaceno para insertarlo.
       */

      if (listInsert(my_list, space) < 0) {
        free(current_word);
        free(space);
      }


    }

    if( fclose(fp) == -1)
      perror("fclose");

    free( paths[j_] );
  }
  free(word_buffer);
}
```

```
/* main
 * --------------
 * Metodo principal de la aplicacion frecpalhilo
 *
 * El enfoque que se dio para hacer posible esta aplicacion fue:
 *
 *  Primero se realiza la busqueda de los archivos a revisar desde el proceso
 *  padre, al comienzo del mismo.
 *
 *  Despues de esto se calcula la reparticion de archivo a los procesos
 *  contadores y se hacen los fork del proceso merger y los procesos contadores.
 *
 *  Counters: Procesos escritores, abren cada archivo recibido, insertan las
 *  palabras encontradas en una lista de frecuencias y escriben la informacion de
 *  las mismas en un pipe no nominal, para que el merger las lea.
 *
 *  Merger: Proceso lector, recibira entrada formateada de un pipe no nominal y
 *  la mezclara en una lista de frecuencias final, que sera presentada de forma
 *  ordenada por stdout.
 *
 *  Por ultimo, en cada proceso se cierran file descriptors del pipe, se liberan
 *  buffers, se hace unlink a semaforos y se cierran.
 */
int main (int argc, char **argv) {
  int n_files, n_ps, quot, rem, aux, i_;
  char **paths ;

  /*  Procesos  */
  int status;

  /*  Pipe  */
  int pipe_fd[2];

  /*  Semaforo  */
  sem_t *mutex, *smp_r, *smp_w;
  int  *r_controller;
  int trash;

  /*  Proceso mezclador  */
  char* word;
  int terminated;
  int* word_size, *frequency;
  list *freq_list;
  node *dummie;

  /*  Procesos contadores   */
  list *my_list;



  /*  Creo pipe */
  if( pipe(pipe_fd)  == -1) {
    fprintf(stderr, "Error abriendo pipe");
    perror("pipe");

    exit(-3);
  }

  /*  Elimino a los semaforos en caso de que existan antes  */
  trash = sem_unlink(SMP0);
```

```
1320  if( trash != 0 && errno != ENOENT)
1321    perror("sem_unlink");
1322
1323  trash = sem_unlink(SMP1);
1324  if( trash != 0 && errno != ENOENT)
1325    perror("sem_unlink");
1326
1327  trash = sem_unlink(SMP2);
1328  if( trash != 0 && errno != ENOENT)
1329    perror("sem_unlink");
1330
1331
1332  /*   Ubico los archivos a procesar    */
1333  paths = (char**) malloc(sizeof(char*) * STANDARD_SIZE); /*perror*/
1334  if (paths == NULL )
1335    perror("malloc");
1336
1337  n_files = myFind(argv[2], &paths);
1338
1339
1340  /*  Calculo el numero de procesos a usar  */
1341  n_ps = MIN(atoi(argv[1]), n_files);
1342
1343
1344  /*  Creo a los semaforos  */
1345  if((mutex=sem_open(SMP0, O_CREAT | O_RDWR, MODE, 1)) == SEM_FAILED){
1346    perror("sem_open");
1347
1348    exit(-2);
1349  }
1350
1351  if((smp_r=sem_open(SMP1, O_CREAT | O_RDWR, MODE, n_ps)) == SEM_FAILED){
1352    perror("sem_open");
1353
1354    exit(-2);
1355  }
1356
1357  if((smp_w=sem_open(SMP2, O_CREAT | O_RDWR, MODE, 0))  == SEM_FAILED){
1358    perror("sem_open");
1359
1360    exit(-2);
1361  }
1362
1363  /*    Proceso merger
1364   *
1365   *  Para empezar cierra los extremos del pipe que no se usaran
1366   *
1367   *  Luego espera por la escritura al pipe en un semaforo
1368   *
1369   *  Del pipe primero lee una variable "r_controller", que indica si
1370   *  un proceso contador ha escrito o si ya termino de escribir.
1371   *  En el primer caso, continua leyendo en el formato de escritura
1372   *  de los procesos contadores.
1373   *
1374   *  Este proceso espera hasta que los contadores hayan dejado de
1375   *  escribir al pipe, y luego imprime una lista de frecuencias
1376   *  ordenada por salida estandar
1377   */
1378
1379  switch( fork() )
```

```
1380  {
1381    default:
1382      break;
1383
1384    case -1:
1385      perror("fork");
1386      fprintf(stderr, "No se pudo abrir el merger\n");
1387
1388      exit(-4);
1389
1390    case 0:
1391
1392      if( close(pipe_fd[WRITE]) == -1)
1393        perror("close");
1394
1395      if( dup2(pipe_fd[READ], 0) == -1)
1396        perror("dup2");
1397
1398      if( close(pipe_fd[READ]) == -1)
1399        perror("close");
1400
1401      word_size = (int *) malloc( sizeof(int) );
1402      if( word_size == NULL)
1403        perror("malloc");
1404
1405      frequency = (int*) malloc(sizeof(int));
1406      if( frequency == NULL )
1407        perror("malloc");
1408
1409      r_controller = (int *) malloc( sizeof(int) );
1410      if( r_controller == NULL )
1411        perror("malloc");
1412
1413      freq_list = (list *) malloc( sizeof(list) );
1414      if( freq_list == NULL )
1415        perror("malloc");
1416
1417
1418      listInit(freq_list);
1419      terminated = 0;
1420
1421      while (terminated != n_ps) {
1422
1423        if( sem_wait(smp_w) == -1) {
1424          perror("sem_wait");
1425          exit(-2);
1426        }
1427
1428        if( sem_wait(mutex)  == -1) {
1429          perror("sem_wait");
1430          exit(-2);
1431        }
1432
1433
1434        read(0, r_controller, sizeof(int) ) ;
1435
1436
1437        if( *r_controller != -1) {
1438          /*Inicializo nodo*/
1439          dummie = (node *) malloc( sizeof(node) );
```

```
1440          if( dummie == NULL ) {
1441            perror("malloc");
1442          }
1443
1444          /*leo el tamano de la palabra*/
1445          read(0, word_size, sizeof(int) );
1446
1447          /*leo la la palabra*/
1448          word = (char *) malloc(*word_size *
1449              sizeof(char) );
1450          if( word == NULL )
1451            perror("malloc");
1452
1453          read(0, word, *word_size + 1 );
1454
1455          /*leo la frecuencia*/
1456          read(0, frequency, sizeof(int) );
1457
1458          nodeInit(dummie, word, *frequency);
1459          listInsert(freq_list, dummie);
1460        }
1461        else
1462          terminated++;
1463
1464
1465        if( sem_post(mutex)  == -1) {
1466          perror("sem_post");
1467          exit(-2);
1468        }
1469
1470        if( sem_post(smp_r)  == -1) {
1471          perror("sem_post");
1472          exit(-2);
1473        }
1474
1475      }
1476
1477      close(0);
1478
1479      free(word_size);
1480      free(frequency);
1481      free(r_controller);
1482
1483      listSort(freq_list);
1484      listPrint(freq_list);
1485
1486      exit(0);
1487  }
1488
1489  /*     Se calcula la reparticion de archivos
1490   *
1491   *  Si se piden mas procesos que archivos encontrados, entonces se utilizaran
1492   *  tantos procesos como archivos.
1493   *
1494   *  De lo contrario se reparte una cantidad de "quot" archivos por proceso,
1495   *  salvo en el ultimo proceso, que recibe "quot + rem" archivos.
1496   *
1497   */
1498
1499  if ( n_ps >= n_files ) {
```

```
    quot = 1;
    rem = 0;
  }
  else {
    quot = n_files / atoi(argv[1]) ;
    rem = n_files % atoi(argv[1]) ;
  }

  /*    Procesos contadores
   *
   *  Luego calculan la frecuencia de las palabras en los archivos
   *  recibidos. Esto se almacena en una lista de frecuencias
   *
   *  Despues de esto pasan a escribir el contenido de la lista en un
   *  pipe, un nodo a la vez (con un formato en particular). Esto es
   *  mediado por semaforo, de forma que mientras un proceso escribe, ninguno
   *  otro lee o escribe.
   *
   *  por ultime liberan buffers y cierran file descriptors
   */

  for(i_=0; i_< n_ps ; i_++) {

    switch( fork() )
    {
      default:
        continue;

      case -1:
        perror("fork");

        exit(-4);

      case 0:

        if ( i_ != n_ps - 1)
          aux = quot;
        else
          aux = quot + rem;


        /*Cierro extremos del pipe no usados*/
        if( close(pipe_fd[READ])  == -1) {
          perror("close");

          exit(-3);
        }

        if( dup2(pipe_fd[WRITE],1)  == -1) {
          perror("dup2");

          exit(-3);
        }

        if( close(pipe_fd[WRITE]) == -1) {
          perror("close");

          exit(-3);
        }

```

```
1560
1561        my_list = (list*) malloc( sizeof(list) );
1562        if (my_list == NULL ) {
1563          perror("malloc");
1564
1565          exit(-1);
1566        }
1567        listInit(my_list);
1568
1569
1570        /*  Inserto nodos y cuento frecuencias  */
1571        countFrequencies(my_list, paths, quot, aux, i_);
1572
1573        /*  Escribo mi lista de frecuencias al pipe */
1574        listPrintRC(my_list, mutex, smp_r, smp_w);
1575
1576
1577
1578        if( sem_close(mutex) == -1)
1579          perror("sem_close");
1580
1581        if( sem_close(smp_r) == -1)
1582          perror("sem_close");
1583
1584        if (sem_close(smp_w) == -1)
1585          perror("sem_close");
1586
1587
1588        close(1);
1589        free(my_list);
1590
1591        exit(0);
1592
1593
1594      }
1595    }
1596
1597    /*  Cierro pipes  */
1598    if( close(pipe_fd[0]) == -1)
1599      perror("close");
1600
1601    if( close(pipe_fd[1]) == -1)
1602      perror("close");
1603
1604    /*  Espero procesos   */
1605    for (i_=0 ; i_<n_ps + 1 ; i_++)   {
1606      if(  wait(&status)  == -1)
1607        perror("waitpid ");
1608    }
1609
1610    /*  Elimino Semaforos */
1611    if( sem_unlink(SMP0) == -1)
1612      perror("sem_unlink");
1613
1614    if( sem_unlink(SMP1) == -1)
1615      perror("sem_unlink");
1616
1617    if( sem_unlink(SMP2) == -1)
1618      perror("sem_unlink");
1619
```

```
  /*  Cierro los semaforos  */
  if( sem_close(mutex) == -1)
    perror("sem_close");

  if( sem_close(smp_r) == -1)
    perror("sem_close");

  if( sem_close(smp_w) == -1)
    perror("sem_close");

  exit(0);
}
--------------------------------------------------------------------------
Archivo: Makefile (frecpalhilo)
OPTS=-ansi -g -Wpedantic -pthread

frecpalhilo: main.o  myFind.o hasht.o singly.o list.o
  gcc ${OPTS} -o frecpalhilo main.o myFind.o hasht.o singly.o list.o -pthread

main.o: main.c myFind.h list.h
  gcc ${OPTS} -c main.c -pthread

myFind.o: myFind.c myFind.h hasht.h
  gcc ${OPTS} -c myFind.c -pthread

hasht.o: hasht.c hasht.h singly.h
  gcc ${OPTS} -c hasht.c

singly.o: singly.c singly.h
  gcc ${OPTS} -c singly.c

list.o: list.c list.h
  gcc ${OPTS} -c list.c

clean:
  rm -rfv *.o freecpalhilo freecpal
--------------------------------------------------------------------------
Archivo: Makefile (frecpalproc)
OPTS=-ansi -Wpedantic

frecpalproc: main.o  myFind.o hasht.o singly.o list.o
  gcc ${OPTS} -o frecpalproc main.o myFind.o hasht.o singly.o list.o -pthread

main.o: main.c myFind.h list.h
  gcc ${OPTS} -c main.c -pthread

myFind.o: myFind.c myFind.h hasht.h
  gcc ${OPTS} -c myFind.c -pthread

hasht.o: hasht.c hasht.h singly.h
  gcc ${OPTS} -c hasht.c

singly.o: singly.c singly.h
  gcc ${OPTS} -c singly.c

list.o: list.c list.h
  gcc ${OPTS} -c list.c

clean:
  rm -rfv *.o frecpalproc
```