TALLER I

Alejandro González - C.I. 4.775.546-2
Sabine Theiss - C.I. 5.781.191-9

1. **Escritura en C++ de todos los tipos de datos correspondientes a las estructuras de datos necesarias para resolver el problema**

Boolean:

```
typedef enum {FALSE, TRUE} Boolean;
```

String:

```
const int MAX = 80;
typedef char = *String;
```

TipoNodo:

```
typedef enum {VALOR, OPERADOR, PARENTESIS} TipoNodo;
```

ValorNodo:

```
typedef struct {int indice;
                TipoNodo discriminante;
                     union      {Boolean valor;
                                 char operador;
                                 char parentesis;
                                 } dato;
                }ValorNodo;
```

ArbolExpre:

```
typedef struct nodoA     {ValorNodo info;
                          nodoA *hizq;
                          nodoA *hder;
                          } NodoA;

typedef NodoA *ArbolExpre;
```

Expresion:

```
typedef struct     {int numero;
                   ArbolExpre arbol;
                   } Expresion;
```

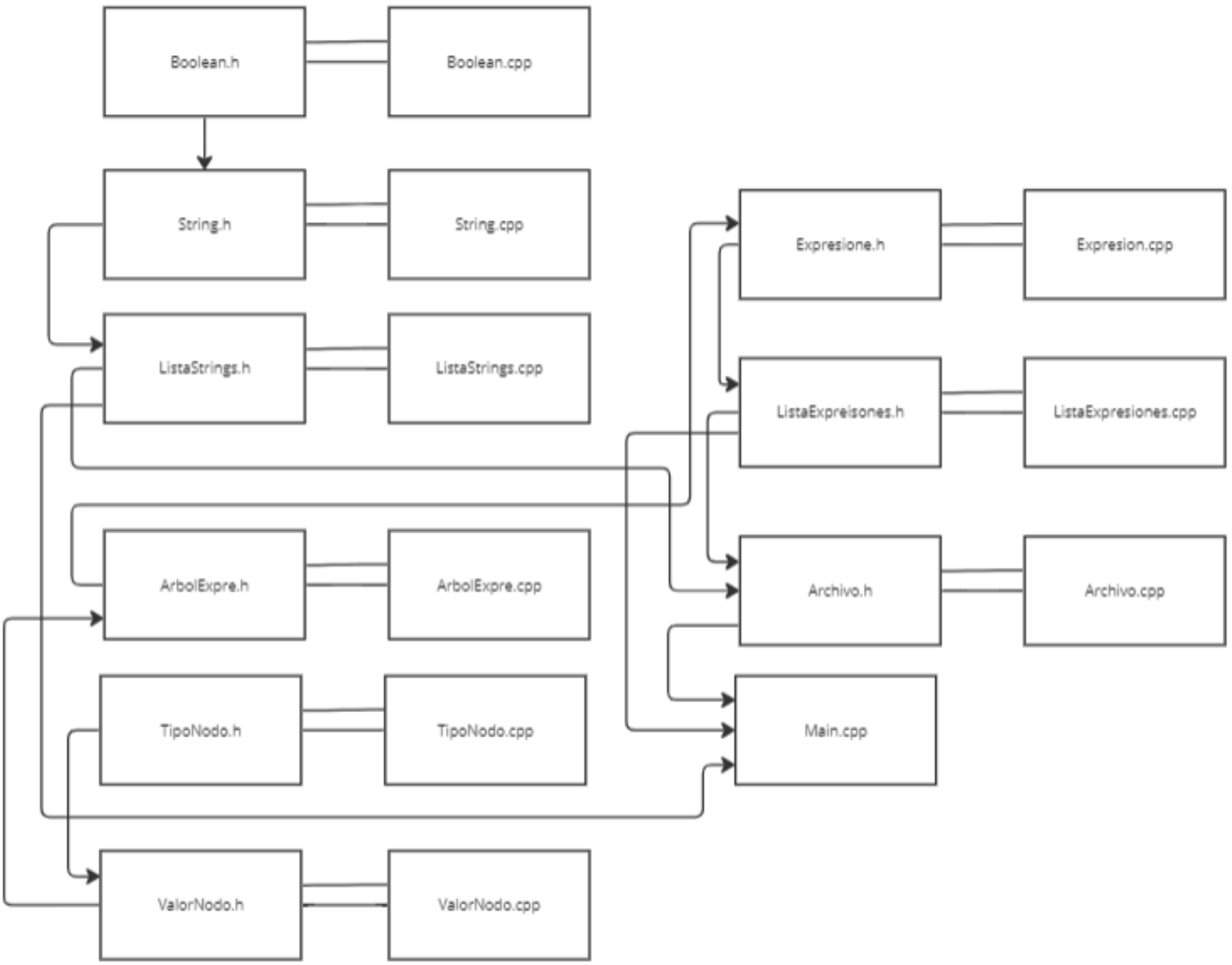
ListaExpresiones:

```
typedef nodoL      {Expresion expre;
                   nodoL *sig;
                   } NodoL;

typedef NodoL *ListaExpresiones;
```

ListaStrings:

```
typedef nodoS      {String palabra;
                   nodoS *sig;
                   }NodoS;

typedef NodoS *ListaStrings;
```

2. **Diagrama de módulos conteniendo los módulos identificados a partir de los tipos de datos definidos, junto con las inclusiones correspondientes**



3. Seudocódigo de la ejecución de cada comando de la aplicación. Dicho seudocódigo debe ser lo más detallado posible, a efectos de permitir luego definir, a partir de él, todos los procedimientos y funciones que sean necesarios.

atomic:

Leer el comando y cargarlo en un string.
Partir el string en varios substrings que se cargarán en una lista de strings. Los espacios en blanco serán los que sirven para hacer la partición en palabras.
Si el primer string de la línea es “atomic”, entonces
    Si el total de palabras de la línea no es 2, entonces
        Mensaje de error: Cantidad de parámetros erróneo
    Sino
        Si la segunda palabra es “true” o “false”, entonces
            Buscar mayor valor en lista de expresiones (el próximo nodo disponible), sumarle uno. Quedarse con ese número como índice de la lista
            Crear nuevo árbol de expresión vacío
            Asignar como tipo “VALOR” el campo “true” (1) o “false” (0) según corresponda, Insertar nuevo nodo como raíz del árbol.
            Crear nuevo nodo en la lista de expresiones, asignar índice de lista, asignar expresión creada como info, insertar nuevo nodo de expresiones en índice de la lista.
            Mostrar mensaje en pantalla “expresión” *índice de lista* “: true” o “false” según corresponda.
        Sino
            Mensaje de error: La segunda palabra debe ser “true” o “false”
        Fin
    Fin
Fin

```
void scan (String &s);
```
← Módulo String

```
void partirStrings (String s, ListaStrings &L);
```
← Módulo Lista Strings

```
int largoListaStrings (ListaStrings L);
```
← Módulo Lista Strings

```
boolean transformarStringABoolean (String s)
```
← Módulo String

```
int mayorIndiceLista (ListaExpresiones LE);
```
← Módulo Lista Expresiones

```
int largoListaExpresiones (ListaExpresiones L);
```
← Módulo ListaExpresiones

```
void crearArbol (ArbolExpre &a);
```
← Módulo ArbolExpre

```
void crearLista (ListaExpresiones &LE);
```
← Módulo Lista de Expresiones

```
void insertarValor (ArbolExpre &a, boolean valor);
```
← Módulo ArbolExpre (un constructor del árbol)

```
void crearExpresion (Expresion &expre, int indice, ArbolExpre arbol)
```
← Módulo Expresion

```
void insertarNodoEnlista (Lista &LE, Expresion e, int numero);
```
← Módulo Lista Expresion

compound

Leer el comando y cargarlo en un string.
Partir el string en varios substrings (palabras) que se cargarán en una lista de strings. Los espacios en blanco serán los que sirven para hacer la partición en palabras.
Si la primer palabra de la línea es “compound”, entonces
    Si el total de palabras de la línea no es 3 o 4, entonces
        Mensaje de error: Cantidad de parámetros errónea, tienen que ser 3 ó 4 palabras.
    Sino
        Ir a la lista de strings, dada una posición, obtener la palabra guardada en posición 2, 3 y 4
        Si el total de palabras de la línea es 3, y la segunda palabra no es “NOT” entonces
            Mensaje de error: NOT debe ser la segunda palabra
        Sino
            Verificar si la tercer palabra de la lista representa un número natural
            Si no representa un número natural, entonces
                Mensaje de error: La tercera palabra debe ser un número natural.
            Sino
                Transformar a número la tercer palabra del contenido
                Buscar en la listaExpresiones si existe alguna expresión cuyo número coincida con ese número de ID.
                Si no existe el número, entonces
                    Mensaje de error: La tercera palabra debe ser un número natural.
                Sino
                    Buscar el árbol correspondiente al número de índice (futuro hijo derecho del árbol)
                    Crear un nuevo árbol,
                    Construir un árbol con nodo raíz NOT, con hijo izquierdo NULL, un hijo derecho que sea el árbol obtenido.
                    Colocar paréntesis (un paréntesis que abre como hoja min a la izquierda del árbol, y un paréntesis que cierra, como hoja max, lo más a la izquierda posible en el árbol).
                    Crear nuevo nodo en la lista de expresiones, asignar índice de lista, asignar expresión creada como info, insertar nuevo nodo de expresiones en índice de la lista.
                    Listar por pantalla el contenido del árbol de expresiones agregado en la lista, en orden.
                Fin
            Fin
        Sino
            Si el total de palabras es 4 y la tercer palabra no es “AND” u “OR”, entonces
                Mensaje de error: Tercer palabra tiene que ser “AND” u “OR”
            Sino
                Verificar si la segunda y la cuarta palabra de la lista representan números naturales
                Si no representan números naturales, entonces
                    Mensaje de error: La segunda y la cuarta cuarta palabra deben ser números naturales.
                Sino
                    Transformar a número la segunda y cuarta palabra del contenido
                    Buscar en la listaExpresiones si existe alguna expresión cuyo número coincida con ese número de ID.
                    Buscar el árbol correspondiente a la palabra 2 (futuro hijo izquierdo del árbol)
                    Buscar el árbol correspondiente a la palabra 4 (futuro hijo derecho del árbol)
                    Crear un nuevo árbol
                    Construir un árbol con nodo raíz tipo operador “AND” u “OR” según corresponda,

Como hijo izquierdo, colocar el árbol obtenido como palabra 2
Como hijo derecho, colocar el árbol obtenido como palabra 4
Colocar paréntesis (un paréntesis que abre como hoja min a la izquierda del árbol, y un paréntesis que cierra, como hoja max, lo más a la izquierda posible en el árbol).
Crear nuevo nodo en la lista de expresiones, asignar índice de lista, asignar expresión creada como info, insertar nuevo nodo de expresiones en índice de la lista.
Listar por pantalla el contenido del árbol de expresiones agregado en la lista, en orden.

Fin

Fin

Fin

Fin

Fin

`void scan (String &s);` ← Módulo String

`void partirStrings (String s, ListaStrings &L);` ← Módulo Lista Strings

`int largoListaStrings (ListaStrings L);` ← Módulo Lista Strings

`string obtenerPalabraDeListaSrings (ListaStrings L)` ← Módulo Lista Strings

`boolean booleanEsNatural (String s);` ← Módulo String

`int transformarANatural (String s)` ← Módulo String

`boolean existeEnListaExpresiones (int numero, ListaExpresiones LE)` ← Módulo Lista Expresiones

`ArbolExpre buscarArbol (ListaExpresiones LE, int numero)` ← Módulo Lista Expresiones

`void copiarArbolAOtro (ArbolExpre ar, ArbolExpre &a)` ← Módulo Árbol Expresiones

`void desplegarNodosArbolOrden (ArbolExpre arbol);` ← Módulo Árbol Expresiones

Constructores

`void cargarParentesis(ArbolExpre &a, char parentesis)` ← Módulo Árbol Expresiones `(crea dos nulos)`

`void cargarOperadorNOT(ArbolExpre &a, ArbolExpre ar, char operador)` ← Módulo Árbol Expresiones

`void cargarOperadorAndOr(ArbolExpre &a, ArbolExpre ar, ArbolExpre arb, char operador)` ← Módulo Árbol Expresiones

`void colocarParentesis (ArbolExpre &a)` ← Módulo Árbol Expresiones `(engancha paréntesis)`

show

Leer el comando de teclado y cargar en un string
Partir el string en varios substrings (palabras), los cuales se cargarán en una lista de strings. Los espacios en blanco en el string original serán los que sirven para hacer la partición
Si la primer palabra de la lista es “show” entonces
    Si el total de palabras en la lista no es 2, entonces
        Mensaje de error: Cantidad de parámetros errónea.
    Sino
        Verificar si la segunda palabra de la lista representa un número natural
        Si no representa un número natural, entonces
            Mensaje de error: La segunda palabra debe ser un número natural.
        Sino
            Transformar a número la segunda palabra del contenido
            Buscar en la lista de expresiones si existe alguna expresión cuyo número coincida con ese número.
            Si no existe la expresión en la lista, entonces
                Mensaje de error: La expresión buscada no existe.
            Sino
                Acceder al árbol de expresión correspondiente
                Recorrerlo recursivamente en órden y listar su contenido por pantalla.
            Fin
        Fin
    Fin
Fin

```
void scan (String &s);
```
← Módulo String

```
void partirStrings (String s, ListaStrings &L);
```
← Módulo Lista Strings

```
boolean streq (String s1, String s2);
```
← Módulo Lista Strings

```
int largo (ListaStrings L);
```
← Módulo Lista Strings

```
boolean esNatural (int numero);
```
← Módulo Lista Strings

```
int transformarANatural (String s)
```
← Módulo Lista Strings

```
boolean existeEnListaExpresiones (ListaExpresiones LE, int numero)
```
← Módulo Lista Expresiones

```
ArbolExpre buscarArbol (ListaExpresiones LE, int numero)
```
← Módulo Lista Expresiones

```
void desplegarNodosArbolOrden (ArbolExpre arbol);
```
← Módulo Árbol Expresiones

```
void print (String s);
```
← Módulo String

evaluate

Leer el comando y cargarlo en un string.
Partir el string en varios substrings (palabras) que se cargarán en una lista de strings. Los espacios en blanco serán los que sirven para hacer la partición en palabras.
Si la primera palabra de la línea es “evaluate”, entonces
Si el total de palabras de la línea no es 2, entonces
Mensaje de error: Cantidad de parámetros errónea
Sino
Si la segunda palabra no es un número natural, entonces
Mensaje de error: La segunda palabra debe ser un natural
Sino
Transformar a número la segunda palabra del contenido
Buscar dentro de la lista de expresiones si existe el número de índice de lista
Si no existe, entonces
Mensaje de error: No existe el número en la lista
Sino
Acceder al nodo correspondiente a la segunda palabra de la lista de expresiones, a la info del nodo
Si el nodo es de tipo “VALOR”, entonces
retornar valor.
Sino
Si valor de nodo es tipo “OPERADOR”, entonces
Si es “A”, entonces
Retornar FuncionRecursiva(hder) &&
FuncionRecursiva(hizq)
Sino
Si es “O”, entonces
Retornar FuncionRecursiva(hder) ||
FuncionRecursiva(hizq)
Sino
Si es ‘N’, entonces
Retornar !FuncionRecursiva(hder)
Fin
Fin
Fin
Fin
Si resultado de la función recursiva vale “true”, entonces
Mostrar por pantalla “la expresión vale true”.
Sino
Mostrar por pantalla “la expresión vale false”.
Fin
Fin
Fin
Fin
Fin

```
void scan (String &s);
```
← Módulo String

```
void partirStrings (String s, ListaStrings &L);
```
← Módulo Lista Strings

```
int largo (ListaStrings L);
```
← Módulo Lista Strings

```
boolean streq (String s1, String s2);
```
← Módulo Lista Strings

```
booleanEsNatural (String s, ListaStrings L);
```
← Módulo Lista Strings

```
int transformarANatural (String s)
```
← Módulo Lista Strings

```
boolean existeEnListaExpresiones (int numero, ListaExpresiones L)
```
← Módulo Lista Expresiones

```
ArbolExpre buscarArbol (ListaExpresiones LE, int numero)
```
← Módulo Lista Expresiones

```
boolean evaluarExpresion(ArbolExpre arbol)
```
← Módulo Árbol de Expresiones

save

Leer el comando y cargarlo en un string.
Partir el string en varios substrings (palabras) que se cargarán en una lista de strings. Los espacios en blanco serán los que sirven para hacer la partición en palabras.
Si la primera palabra de la línea es “save”, entonces
Si el total de palabras de la línea no es 3, entonces
Mensaje de error: Cantidad de parámetros errónea
Sino
Si la segunda palabra no es un número natural, entonces
Mensaje de error: La segunda palabra debe ser un natural
Sino
Si la tercera palabra no es un string, y no termina en “.dat”
Mensaje de error: La tercera palabra debe ser un string, y el tipo de archivo debe ser .dat
Sino
Transformar a número la segunda palabra del contenido
Buscar en la lista de expresiones si existe alguna expresión cuyo número coincida con ese número.
Si no existe la expresión en la lista, entonces
Mensaje de error: La expresión buscada no existe.
Sino
Verificar si el archivo ya existe, entonces
Si el archivo no existe, entonces
Asignar un entero a cada nodo del árbol en órden
Abrir archivo
Recorrer el árbol en preorden, bajándolo a archivo con su respectivo número de
Cerrar archivo
Sino
Preguntar al usuario si desea sobreescribirlo. Admitir “S” o “N” como respuesta
Si usuario ingresa “S”, entonces
Asignar un entero a cada nodo del árbol en orden.
Abrir archivo
Recorrer el árbol en preorden, bajándolo a archivo con su respectivo
Mostrar en pantalla el siguiente mensaje:
“expresión *número de expresión* respaldada correctamente en *nombrearchivo*.dat”
Cerrar archivo
Sino
Si usuario ingresa “N”, entonces
Mensaje apropiado de salida
Sino
Mensaje de error: Por favor ingresar solamente “S” o “N”
Fin
Fin
Fin
Fin
Fin
Fin
Fin

```
void scan (String &s);
```
← Módulo String

```
void partirStrings (String s, ListaStrings &L);
```
← Módulo Lista Strings

```
int largo (ListaStrings L);
```
← Módulo Lista Strings

```
boolean streq (String s1, String s2);
```
← Módulo Lista Strings

```
booleanEsNatural (String s, ListaStrings L);
```
← Módulo Lista Strings

```
int transformarANatural (String s)
```
← Módulo Lista Strings

```
boolean existeArchivo (String nombrearchivo);
```
← MóduloArchivo

```
bajarEvaluacion (String nombrearchivo, file *f);
```
← MóduloArchivo

```
void enumarNodos (ArbolExpre &a)
```
← Módulo Árbol Expresiones

load

Leer el comando y cargarlo en un string.
Partir el string en varios substrings (palabras) que se cargarán en una lista de strings. Los espacios en blanco serán los que sirven para hacer la partición en palabras.
Si la primera palabra de la línea es “load”, entonces
    Si el total de palabras de la línea no es 2, entonces
        Mensaje de error: Parámetros incorrectos, deben de ser dos palabras
    Sino
        Si la segunda palabra no es un string, y no termina en “.dat”
        Mensaje de error: Se debe ingresar un string (solo caracteres) como segunda palabra y debe terminar en .dat
        Sino
            Verificar si el archivo existe
            Si no existe, entonces
                Mensaje de error: Archivo no existe
            Sino
                Buscar mayor valor en lista de expresiones (el próximo nodo disponible), sumarle uno.
                Quedarse con ese número como índice de la lista.
                Abrir archivo
                Leer empezando desde el principio los estructurados guardados en el archivo
                Crear un nuevo árbol de expresión, insertar secuencialmente los nuevos nodos numerados hasta llegar al final del archivo.
                La inserción se basa en el criterio de un ABB.
                Cerrar archivo.
                Crear nuevo nodo en la lista de expresiones, asignar índice, asignar el árbol nuevo, insertar nuevo nodo de lista al final de la misma (insback).
                Mostrar por pantalla: “expresion 1:”
                Recorrer el árbol recursivamente en órden y listar su contenido por pantalla.
            Fin
        Fin
    Fin
Fin

```
void scan (String &s);
```
← Módulo String

```
void partirStrings (String s, ListaStrings &L);
```
← Módulo Lista Strings

```
int largo (ListaStrings L);
```
← Módulo Lista Strings

```
boolean streq (String s1, String s2);
```
← Módulo Lista Strings

```
boolean ExisteArchivo (String nombrearchivo);
```
← MóduloArchivo

```
levantarEvaluacion (String &nombrearchivo, file *f);
```
← MóduloArchivo

```
ArbolExpre buscarArbol (ListaExpresiones LE, int numero)
```
← Módulo Lista Expresiones

```
void insertarDeArchivoEnOrden (Expresion &expre, int indice)
```
← Módulo Expresion

```
void desplegarNodosArbolOrden (ArbolExpre arbol);
```
← Módulo Árbol Expresiones

```
void print (String s);
```
← Módulo String

exit

Leer el comando y cargarlo en un string.

Partir el string en varios substrings (palabras) que se cargarán en una lista de strings. Los espacios en blanco serán los que sirven para hacer la partición en palabras.

Si la primera palabra de la línea es “exit”, entonces

- Si el total de palabras de la línea no es 1, entonces
  - Mensaje de error apropiado
- Sino
  - Recorrer la lista de expresiones entrando en cada árbol de expresiones, liberando toda la memoria dinámica usada
  - Recorrer la lista de strings, liberando toda la memoria dinámica usada
  - Liberar strings de nombres de archivos
  - Mostrar mensaje por la pantalla “hasta la proxima”
- Fin

```
void scan (String &s);
```
← Módulo String

```
void partirStrings (String s, ListaStrings &L);
```
← Módulo Lista Strings

```
int largo (ListaStrings L);
```
← Módulo Lista Strings

```
void liberarMemoriaListaE (ListaExpresiones &LE);
```
← Módulo Lista Expresiones

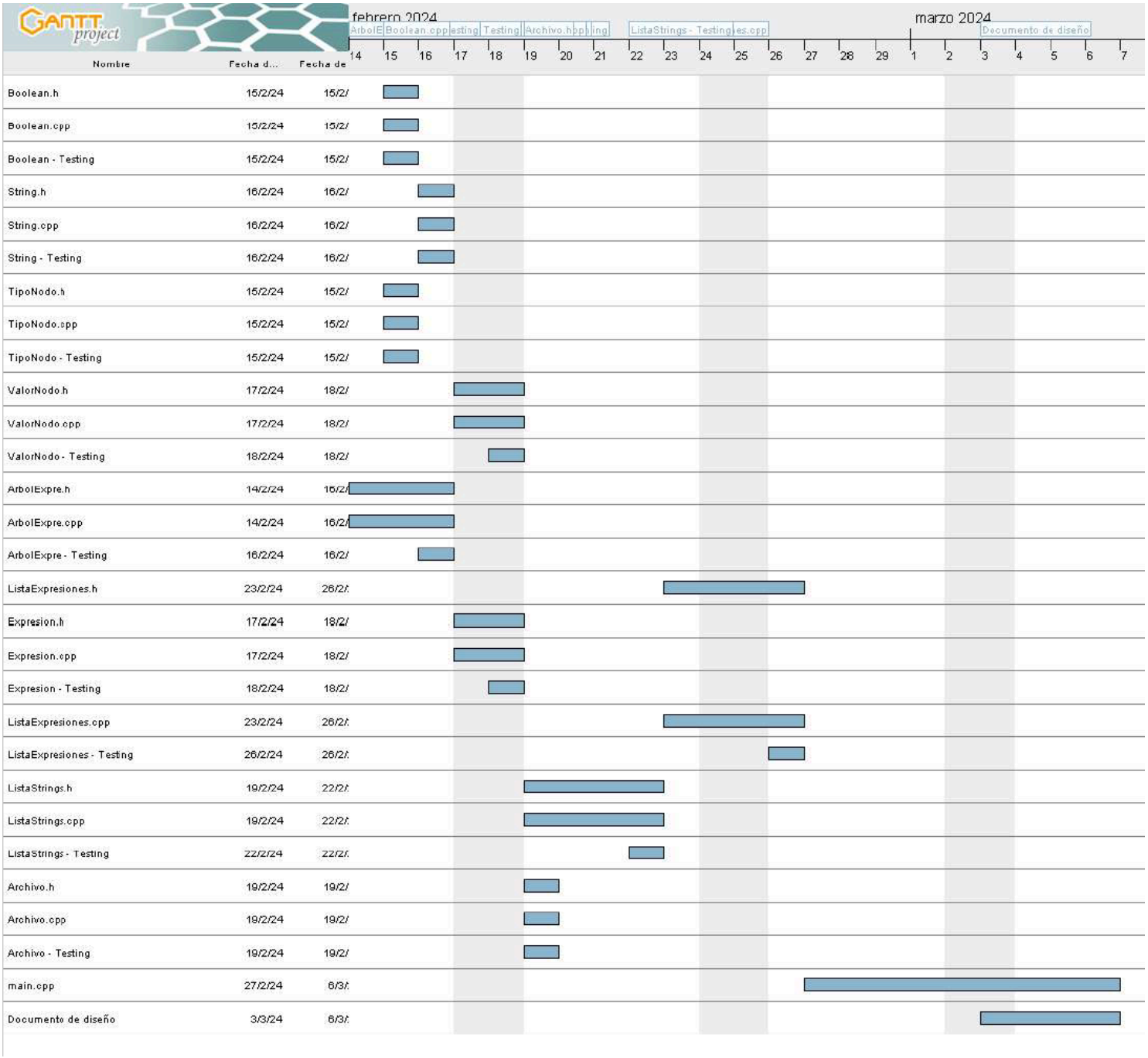
```
void liberarMemoriaArbol (ArbolExpre &arbol);
```
← Módulo Árbol Expresiones

```
void liberarMemoriaListaS (ListaStrings &L);
```
← Módulo Lista Strings

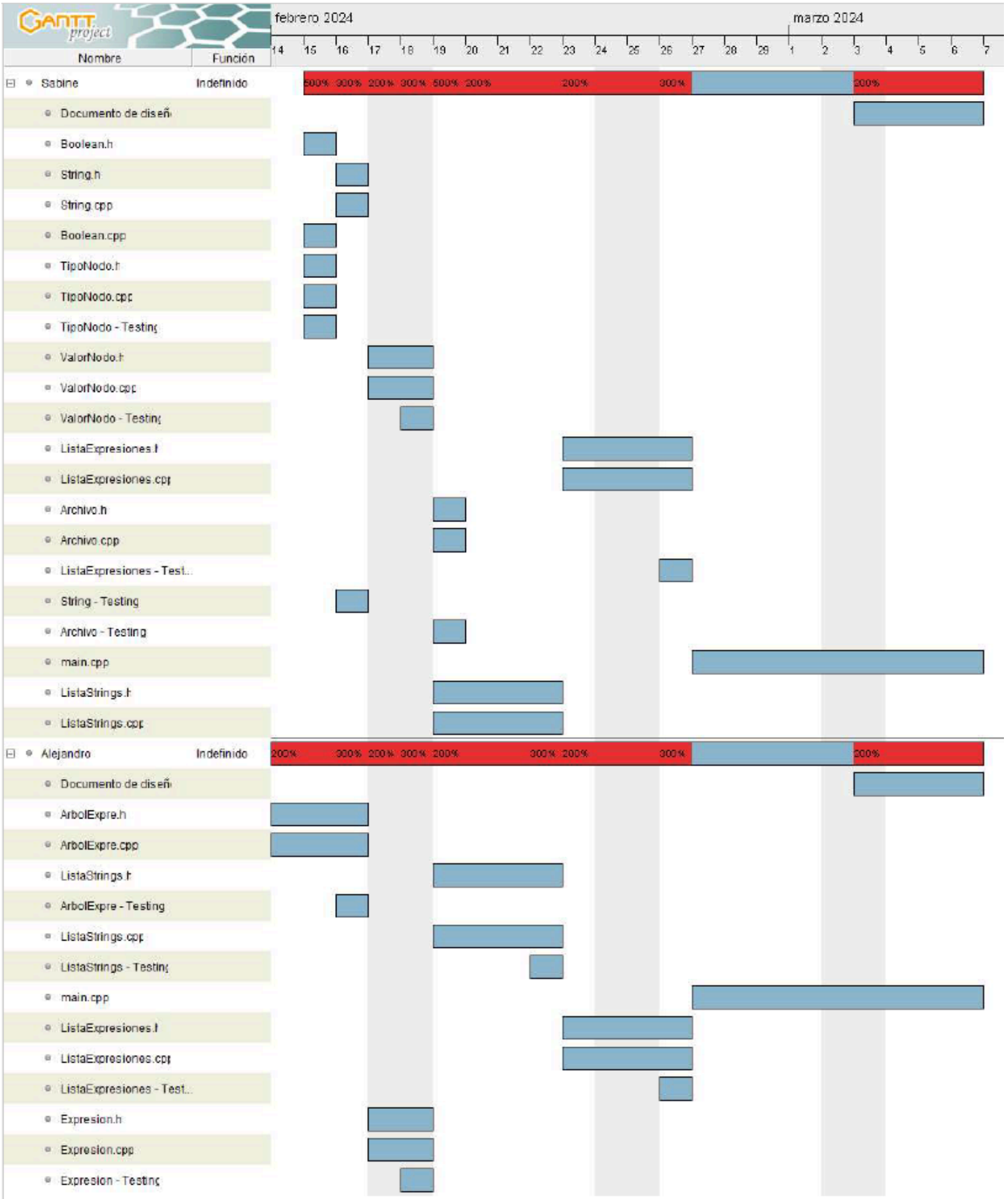
```
strdestruir(nombrearchiv);
```
← Módulo Strings

5. Cronograma de implementación y testing de los módulos indicando, para cada módulo a implementar y testear, fechas estimadas de inicio y de fin

Vista general:



Vista por personal:



6. Explicación de todas las decisiones de diseño que haya ido tomando el equipo al realizar el diseño de la solución.

Estructura de datos

Se adaptaron las estructuras planteadas sugeridas, lo cual incluye:

- Un tipo de enumerado para representar el valor booleano “TRUE” o “FALSE”, según la estructura utilizada en cursos anteriores.
- Un string dinámico (puntero a espacio de memoria asignada en tiempo de ejecución). Se optó por esta estructura por ser la más reciente utilizada en el curso para almacenar strings y porque tiene la capacidad de generar memoria dinámica en tiempo de ejecución.
- Para cada expresión se utilizará un árbol binario. En cada nodo del árbol se guarda un estructurado el cual va a contener en el caso de ser un paréntesis, un caracter de tipo char, en caso de ser un valor booleano un enumerado y en caso de contener un operador (AND, OR o NOT) un caracter (‘A’, ‘O’ u ‘N’). Para distinguir entre los diferentes tipos (PARENTESIS, OPERADOR, VALOR), se usa una unión discriminada.
  Además contendrá un tipo entero que indicará el índice, el cual se bajará en caso de guardar una expresión en archivo; al levantarse se utilizará para insertar los nodos ordenados de menor a mayor.
  Se utiliza esta estructura, porque el contenido a almacenar no tiene cota máxima y se tienen que poder almacenar datos de diferente tipo, pudiendo conectar dos ramas diferentes según el dato ingresado.
- Cada expresión formará parte de un estructurado que contiene un número de índice por el cual se identificará el árbol de expresión correspondiente dentro de la lista y su contenido. Lo mismo es necesario para poder ordenar el contenido en órden cronológico.
- Las expresiones irán en una lista enumerada de expresiones que quedará ordenada según el índice de cada estructurado. Se utiliza esta estructura porque no hay cota máxima y los datos a almacenar son de tipo estructurado.
- Para almacenar las palabras ingresadas por el usuario se utilizará una lista de strings dinámicos. Se usa este tipo de estructura porque no hay cota máxima y se quiere guardar un tipo complejo (puntero a un string dinámico).

Decisiones de diseño de algoritmos

Atomic

Para los casos en que el usuario ingresa el comando atomic, desarrollamos un algoritmo que verifica en primer lugar que no existan errores en el input que ingresó el usuario (debe ser de forma `atomic valor`). En caso de no existir errores, se crea un nuevo árbol vacío, se crea un nuevo nodo raíz para el árbol, y se le asigna un valor TRUE o FALSE. En la lista de expresiones se crea un nuevo nodo para guardar el nuevo árbol, este nuevo nodo se crea con un índice que represente la cantidad de nodos actual de la lista más uno. Finalmente se imprime por pantalla un mensaje al usuario indicando en qué número de índice de la lista se guardó la nueva expresión atomic.

Compound

Cuando el usuario ingresa el comando compound, el algoritmo verifica primero que la cantidad de parámetros ingresados sea correcta (`compound expre`), pudiendo ser “expre” del formato `(exp1 AND exp2)`, `(exp1 OR exp2)` o `(NOT exp1)`. Deben ser tres en el caso de que el operador sea NOT, y cuatro en el caso de que el operador sea AND u OR.

Para los casos NOT, se verifica que la tercera palabra ingresada por el usuario sea un número natural, si lo es, esa palabra se transforma a número, y luego se busca en la lista de expresiones si existe ese número de índice.
Si el número de índice existe, se busca el árbol que le corresponda, este árbol será más adelante insertado como hijo derecho del árbol principal.

El paso siguiente será crear un árbol vacío, y a ese árbol le insertamos un nodo raíz que contenga un NOT, un hijo izquierdo que sea NULL, y un hijo derecho que sea el árbol obtenido anteriormente.
Luego, completamos el árbol colocando los paréntesis, creamos un nuevo nodo en la lista de expresiones, y le insertamos el nuevo árbol creado.
Por último, listamos por pantalla el contenido del árbol de expresiones en orden.

Para los casos AND u OR, se verifica que la segunda y la cuarta palabra ingresadas por el usuario representen números naturales, si lo son, esas palabras se transforman a número, y luego se busca en la lista de expresiones si existen esos números de índice.
Se busca en la lista de expresiones el árbol correspondiente a la palabra 2, ese árbol será el futuro hijo izquierdo de nuestro árbol principal.
Se busca en la lista de expresiones el árbol correspondiente a la palabra 4, ese árbol será el futuro hijo derecho de nuestro árbol principal.
El paso siguiente consiste en crear un nuevo árbol, se le crea un nodo raíz que será un operador AND u OR según corresponda, y luego Como hijo izquierdo, se coloca el árbol obtenido como palabra 2, y como hijo derecho, se coloca el árbol obtenido como palabra 4.
Luego se completa el árbol colocando los paréntesis de apertura y cierre, creamos un nuevo nodo en la lista de expresiones, y le insertamos el nuevo árbol creado.
Por último, listamos por pantalla el contenido del árbol de expresiones en orden.

Show

En el caso que el usuario ingrese “show”, se verifica si la cantidad de palabras es la correcta (2), ya que el formato admitido es `show n`. En el caso que no sea sí, se emite un mensaje de error y termina la ejecución. Luego si la segunda palabra (contando el show, es decir la que le sigue al show) es un número natural (ya que la forma del comando dicta que así sea). Para chequear esto, se hará una función que revise si un string ingresado es un natural. En el caso de que no se trate de un natural, termina la ejecución con un mensaje de error adecuado. En el caso que lo sea, se transforma el número (actualmente un string) en un número natural. Para esto utilizaremos una función que transforme un string a un número natural.
Luego se busca en la lista de expresiones si existe una expresión cuyo número (índice) coincida con ese natural. Si no existe ninguno, se termina la ejecución con un mensaje de error apropiado. De lo contrario, se accede al árbol de expresión correspondiente y se recorre en órden, desplegando por pantalla su contenido mediante un procedimiento que lista el contenido del árbol por pantalla..

Evaluate

Este algoritmo primero evalúa si el total de palabras ingresadas por el usuario es de 2 (ya que la forma de ingresar el comando es `evaluate n`, de lo contrario se sale de la ejecución con un mensaje de error apropiado.
Después el algoritmo valida si la segunda palabra ingresada es un número natural (ya que la forma del comando así lo dicta). Se utilizará una función para determinar si la palabra es un número natural. Si no lo es, termina la ejecución con un mensaje de error adecuado.
De lo contrario se transforma la palabra en un natural mediante una función que lo convierta en natural y se busca en la lista de expresiones si existe una expresión cuyo número (índice) coincida con ese natural. Si no existe ninguno, se termina la ejecución con un mensaje de error apropiado.

De lo contrario, se accede a ese lugar de la lista al árbol correspondiente. Se ejecuta una función recursiva que siempre que la info del nodo sea del tipo “VALOR”, retorne su contenido (TRUE o FALSE).

Si es de tipo “OPERADOR”, se ejecuta la misma función sobre el hijo derecho y el hijo izquierdo, en el caso de “AND” retornando el resultado del hijo izquierdo && hijo derecho, en el caso del “OR” retornando el resultado del hijo izquierdo || hijo derecho, y en el caso del “NOT, retornando el resultado de !del hijo derecho solamente (ya que el árbol en el caso del NOT nunca tendrá otra cosa que un paréntesis en el hijo izquierdo).

Los paréntesis se ignoran en la función recursiva.
Luego de la función recursiva, mostramos por pantalla “expresión vale true” en el caso de que el resultado de la función sea true y “expresión vale false” si el resultado de la función es false.

Save

Este algoritmo valida primero si el total de palabras ingresadas por el usuario es 3, ya que la sintaxis del comando es `save salve n nombrearchivo.dat`. Si el usuario ingresó menos o más palabras, mostramos un mensaje de error y se termina la ejecución.
De lo contrario, se valida si la segunda palabra es un número natural, ya que la síntaxis debe ser de esa forma. Para lo mismo utilizaremos una función que determine si un string ingresado es un número natural. Si no es un natural, se emite un mensaje de error y se sale de la ejecución.
Si no, se transforma la segunda palabra en un natural utilizando una función que transforme un string a natural.
Si la tercer palabra no es un string y no termina en “.dat” (como la sintaxis dicta), se emite un mensaje de error y se sale de la ejecución.
De lo contrario, se transforma a natural la segunda palabra mediante una función que transforme un string a natural.
Próximo, se verifica si existe en la lista de expresiones alguna expresión cuyo número coincida con ese número. Si no hay, se emite un mensaje de error y se termina la ejecución.
De lo contrario, se verifica si el archivo ya existe (mediante una función que lo determine).
Si el archivo ya existe, se preguntará al usuario si lo quiere sobreescribir, admitiendo “S” o “N” como posibles respuestas (si no, se mostrara un error que solo se pueden ingresar esas letras). Si el usuario ingresa “S”, y también en el caso de que el archivo no haya existido antes, Si enumerán los nodos del árbol en órden, asignando un entero a campo “numero” de ValorNodo.
Abrir el archivo y cargarle el contenido de la expresión (el índice del nodo y de su info el contenido de la unión, es decir, “C”, “A”, “O”, “N”, true o false), recorrdiendo el árbol en preorden (primero la raiz, luego hijo izquierdo, luego hijo derecho).
Cuando se termine el archivo, se termina la recorrida y se cierra el archivo.

Load

Cuando el usuario ingresa el comando load, el algoritmo verifica primero que la cantidad de palabras ingresadas sea correcta, deben ser dos (porque la síntaxis es `load nombrearchivo.dat`.) También se verifica que la segunda palabra sea un string, y que finalice en la extensión “.dat”. Si alguna de estas condiciones no se cumple, se emite un mensaje de error.
Cuando no hay errores en el ingreso de los parámetros, se sigue adelante verificando que exista un archivo con ese nombre. Si no existe, se muestra un mensaje de error adecuado. Si el archivo existe, se debe abrir. A su vez, se debe tomar el número de índice de lista, crear un nuevo árbol de expresión para poder insertar a lo levantado del archivo en los nodos, ordenado por los índices de menor a mayor (en órden) hasta llegar al final del archivo (como si fuera un ABB).
Se debe cerrar el archivo.
Se crea un nuevo nodo en el índice de la lista de expresiones y se asigna el nuevo nodo al final de la lista.
Al final de muestra por pantalla “expresion 1:” y se recorre el nuevo árbol en órden mostrando por pantalla el contenido.

Exit

Cuando el usuario ingresa el comando exit, se verifica que la cantidad de palabras ingresadas sea solamente una (ya que la sintáxis es `exit`. Si es así, se recorre la lista de expresiones liberando toda la memoria dinámica utilizada. Asimismo, se recorre la lista de strings, liberando también toda la memoria dinámica utilizada. Se liberan también los strings de nombres de archivos, y finalmente se emite un mensaje por pantalla para el usuario “hasta la próxima”.