# TALLER I

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

# Indice


**1. Estructura de datos** **3**
- Decisiones de estructura de datos 5
- Decisiones de diseño de algoritmos 5
  - atomic 5
  - compound 6
  - show 7
  - evaluate 7
  - save 8
  - load 9
  - exit 9

**2. Diagrama de módulos** **10**

**3. Pseudocódigo de la ejecución de cada comando de la aplicación** **11**
- atomic 11
- compound 14
- show 19
- evaluate 22
- save 25
- load 29
- exit 33

**4. Cronograma de implementación y testing** **35**
- Vista general: 35
- Vista por persona: 36

**5. Cronograma luego de la implementación** **37**
- Vista general: 37
- Vista por persona: 38

**6. Proceso de creación del programa, problemas enfrentados y soluciones encontradas** **40**
- Booleano 40
- String 40
- TipoNodo 40
- ValorNodo 40
- Archivo 40
  - Proceso de bajar árbol a archivo: 41
- ArbolExpre 42
- Expresion 42
- ListaStrings 42
- ListaExpresiones 43
- main 44

**7. Balance de trabajo y conclusiones del taller en general** **44**

## 1. Estructura de datos

**Boolean:**

```
typedef enum {FALSE, TRUE} boolean; // cambiamos a "b"
```

**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 struct nodoL     {Expresion expre;
                         nodoL *sig;
                         } NodoL;

typedef NodoL *ListaExpresiones;
```

**ListaStrings:**

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

typedef NodoS *ListaStrings;
```

**Errores:**

```
typedef enum {
            COMANDOENLUGAREQUIVOCADO,NOCONTIENEFORMATO,
            PALABRAENLUGAREQUIVOCADO,CANTIDADDEPARAMETROSINCORRECTA,PALABRAEQUIVOCADA,
            PALABRADEBESERNATURAL, CUARTAPALABRADEBESERNATURAL, NOEXISTEENLISTAEXPRESIONES,
            NOEXISTEENLISTAEXPRESIONESPRIMERO, NOEXISTEENLISTAEXPRESIONESSEGUNDO,
            PALABRANOALFABETICA, ARCHIVONODAT, NOEXISTEARCHIVO
            }errores;
```

**Decisiones de 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. Además resulta más rápido recorrer el contenido de un árbol, pudiendo discriminar por el índice en el caso de tenerlo asignado.
- 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. Se utiliza un estructurado porque contiene datos de diferente tipo (el árbol y el entero índice).
- 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 puntero a un string dinámico. Al tener las palabras divididas en nodos, facilita la búsqueda por palabras.
- Para el manejo de errores se utiliza un tipo enumerado. Se optó por ese tipo para poder discriminar por nombre según el error de manera ordenada.

## 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 el valor TRUE o FALSE (transformando String a boolean).
En la lista de expresiones se crea un nuevo nodo para guardar el nuevo árbol, a este nuevo nodo se le asigna un índice que represente la cantidad de nodos actual (largo) de la lista más uno. Finalmente se muestra por pantalla un mensaje al usuario indicando en qué número de índice de la lista se guardó la nueva expresión atomic junto a la expresión.

### 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. También se revisa si el orden es correcto.

Para los casos NOT, se verifica que la tercera palabra ingresada por el usuario sea un número natural (con función atoi() de la biblioteca stdlib), si lo es, esa palabra se transforma a número (utilizando también atoi())), 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 la 'N' (String "NOT" transformado a char) y los dos hijos vacíos en principio. Luego en su hijo derecho se copia el árbol obtenido anteriormente. Finalmente insertamos un paréntesis de apertura en el mínimo del árbol entero y un paréntesis de cierre en el máximo del árbol entero.
Se asigna el árbol al miembro del estructurado Expresion, se asigna como su índice el largo de la lista expresiones más uno.
Se crea un nuevo nodo en la lista de expresiones, y se insierta la nueva expresión.
Por último, listamos por pantalla el contenido de ese lugar de la lista (todos los nodos del árbol en cuestión 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. Si existen, se discrimina si la tercer palabra es "AND" y "OR" y lleva a cabo la operación de carga del árbol según esa palabra:
Se busca en la lista de expresiones el árbol correspondiente a la palabra 2, ese árbol será el futuro hijo izquierdo del nuevo árbol principal.

Se busca en la lista de expresiones el árbol correspondiente a la palabra 4, ese árbol será el futuro hijo derecho del nuevo árbol principal.
El paso siguiente consiste en crear un nuevo árbol, se le crea un nodo raíz que será 'A' o 'N' (operador transformado a char correspondiente) según corresponda, e inicialmente los dos hijos vacíos. Luego en el hijo izquierdo, se copia el árbol obtenido como palabra 2, y como hijo derecho, se copia el árbol obtenido como palabra 4.
Finalmente insertamos un paréntesis de apertura en el mínimo del árbol entero y un paréntesis de cierre en el máximo del árbol entero.
Se asigna el árbol al miembro del estructurado Expresion, se asigna como su índice el largo de la lista expresiones más uno.
Se crea un nuevo nodo en la lista de expresiones, y se insierta la nueva expresión.
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 (2) y el orden es correcto, 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 (atoi() de la biblioteca stlib)).
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 ejecuta un procedimiento que siempre que sea igual el número al número del nodo de ListaExpresiones, se despliega la expresión correspondiente: Se selecciona su id (número de expresión) y se muestra por pantalla "expresion" y el id y se ejecuta el procedimiento recursivo en orden sobre el árbol de expresión correspondiente, desplegando por pantalla su contenido (en el despliegue de información se hace uso de otro procedimiento de ValorNodo que discrimina y ejecuta el desplegado correspondiente de la información del nodo según si es VALOR, OPERADOR o PARENTESIS, llamando a las operaciones de desplegado correspondiente).

**evaluate**

Este algoritmo, si es el caso que "evaluate" pertenece a ListaStrings, evalúa si el el comando aparece en el primer lugar, luego 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) con la función atoi() de stlib. 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 (también con atois()) 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 que siempre que la info del nodo sea del tipo “VALOR”, retorne su contenido (se retorna el booleano TRUE o FALSE).

Si es de tipo “OPERADOR”: En el caso que el operador sea ‘N’ (NOT), se ejecuta la misma función sobre su hijo derecho precedido de ! y se retorna el booleano “TRUE” si vale true y “FALSE” si vale false.
En caso que el operador sea ‘A’ (AND), se ejecuta la misma función sobre el hijo izquierdo seguido de && y se ejecuta la misma función sobre el hijo derecho, retornando el resultado, si el resultado vale true, se retorna el booleano “TRUE” y si vale false, el booleano “FALSE.
En el caso que el operador sea ‘O’ (OR), se ejecuta la misma función sobre el hijo izquierdo seguido de || y se ejecuta la misma función sobre el hijo derecho, retornando el resultado, si el resultado vale true, se retorna el booleano “TRUE” y si vale false, el booleano “FALSE.
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 si “save” está en primer lugar de ListaStrings, si no lo está, se muestra un mensaje de error. Si el total de palabras ingresadas por el usuario es 3, ya que la sintaxis del comando es `save 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 (con atoi() de biblioteca stlib). 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 natura (también con atoi())l.
Dividimos la tercer palabra en nombre del archivo y la extensión con un procedimiento parecido a dividirString(), pero que usa “.” como indicador de separación de palabras en vez del espacio en blanco.
Si el nombre del archivo no es alfabético o la extensión 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 (otra vez con atoi()), ese será el Id.
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 correspondiendo a la tercer palabra ya existe (mediante un while() que solo se sale mientras la letra ingresada no es igual a “S” o a “N”).
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, se selecciona el árbol dentro de la expresión correspondiente al número de Id y se enumerán los nodos del árbol en órden, asignando un índice al campo “indice” de ValorNodo. Esto se hace mediante un procedimiento recursivo al cual se le pasa el índice (seteado afuera del procedimiento inicialmente en 0) y el árbol por referencia. Se recorre con el mismo procedimiento el subarbol izquierdo, se suma uno al índice y se llama al procedimiento de ValorNodo de asignarIndice en la raíz, luego se recorre el subarbol derecho con el mismo procedimiento, así enumerando todos los nodos en orden.
Abrir el archivo con el nombre correspondiente a la tercer palabra y bajar el árbol correspondiente al Id mediante un procedimiento que baja ValorNodo (el índice del nodo, su discriminante y de su info el contenido de la unión discriminando según su tipo, es decir, “C”, “A”, “O”, “N”, TRUE o FALSE)), recorrdiendo el árbol en preorden (primero la raiz, luego hijo izquierdo, luego hijo derecho).
Se cierra el archivo y se muestra por mensaje “expresion” *Id* respaldada correctamente.

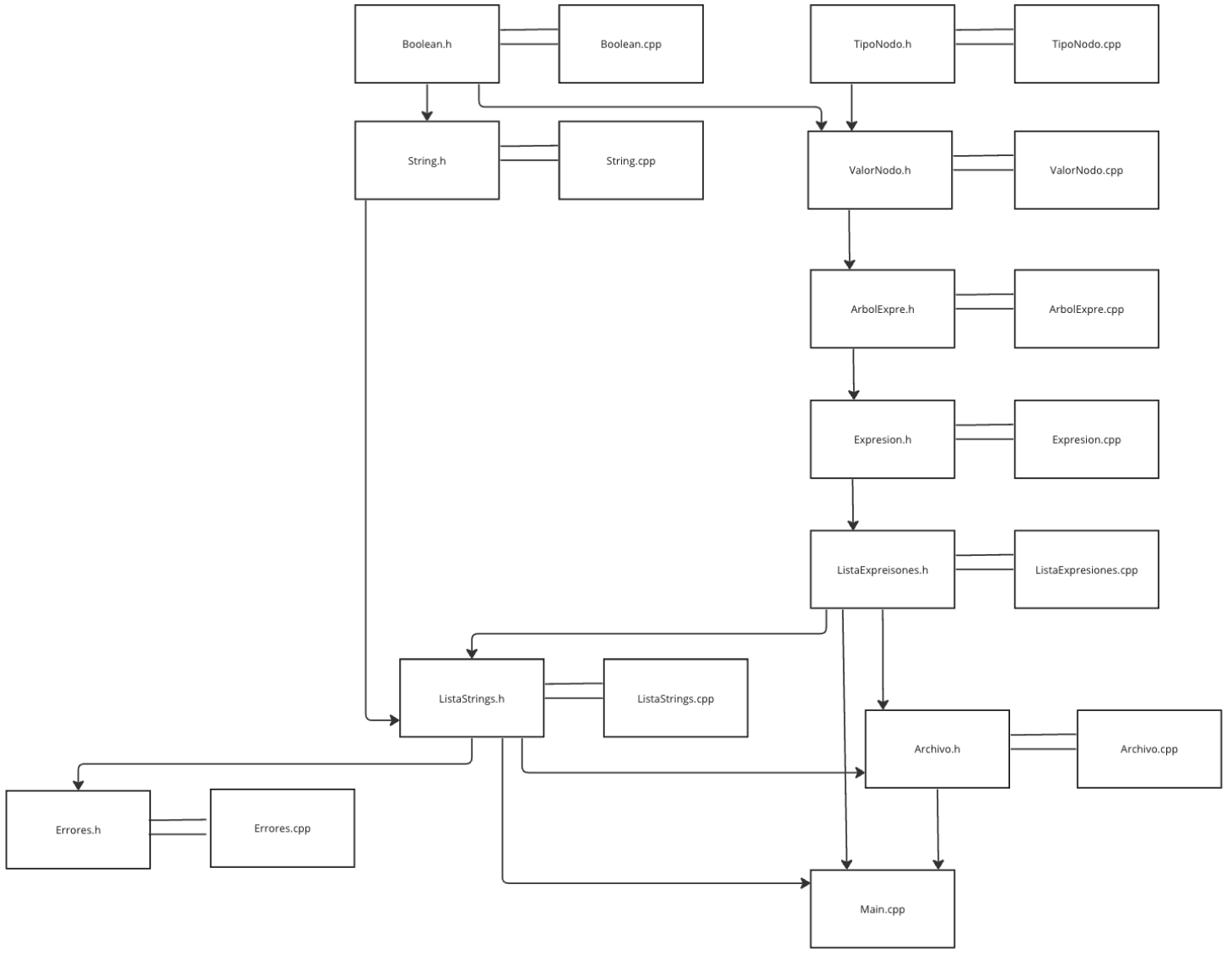
**load**

Cuando el usuario ingresa el comando load, el algoritmo verifica si “load” se encuentra en la primer posición de ListaStrings, si no se muestra un mensaje de error y se sale de la ejecución. Sino, se verifica que la cantidad de palabras ingresadas sea correcta, sean dos (porque la síntaxis es `load nombrearchivo.dat`.), de lo contrario muestra mensaje de error y se sale de la ejecución.
Dividimos la segunda palabra en nombre del archivo y la extensión con un procedimiento parecido a dividirString(), pero que usa “.” como indicador de separación de palabras en vez del espacio en blanco.
Si el nombre del archivo no es alfabético o la extensión no termina en “.dat” (como la sintaxis dicta), se emite un mensaje de error y se sale de la ejecución.
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 ejecuta el procedimiento de levantarArbol que pasa árbol por referencia. Lo que hace el procedimiento es, abrir el archivo, levantar ValorNodo (el primer índice del archivo, la primer discriminante y primer operador, valor o paréntesis discriminando según su tipo). Luego, mientras no se haya llegado al final del archivo, se llama al procedimiento insertarValorEnOrden() que siempre que el árbol esté vacío, inserta un nuevo nodo en la raíz. Eso va a pasar con el primer ValorNodo leído desde el archivo. De lo contrario, si el índice leído es menor de lo que se encuentra en el nodo del árbol sobre el cual se está parado, se realiza al mismo procedimiento sobre su hijo izquierdo, de lo contrario se llama al mismo procedimiento sobre su hijo derecho. De esta forma se insertará el contenido del árbol de menor a mayor según sus índices, como si fuera un ABB.
Si levante el próximo ValorNodo y se sigue así hasta que se termine el archivo.
Se cierra el archivo
Luego, se asigna como NumeroExpresion el largo de ListaExpresiones existente más uno, se asigna el árbol al miembro de Expresión, Se crea un nuevo nodo en el índice de la lista de expresiones y se asigna el nuevo nodo al final de la lista.
Se ejecuta un procedimiento que siempre que sea igual el número al número del nodo de ListaExpresiones, se despliega la expresión correspondiente: Se selecciona su id (número de expresión) y se muestra por pantalla “expresion” y el id y se ejecuta el procedimiento recursivo en orden sobre el árbol de expresión correspondiente, desplegando por pantalla su contenido (en el despliegue de información se hace uso de otro procedimiento de ValorNodo que discrimina y ejecuta el desplegado correspondiente de la información del nodo según si es VALOR, OPERADOR o PARENTESIS, llamando a las operaciones de desplegado correspondiente).

**exit**

Cuando el usuario ingresa el comando exit, se verifica si “exit” es la primer palabra ingresada, de lo contrario se muestra un mensaje de error y se sale de la ejecución. Luego se verifica que la cantidad de palabras ingresadas sean solamente una (ya que la sintáxis es `exit`. Si es así, se recorre la lista de expresiones liberando toda la memoria dinámica utilizada. Esto se realiza mediante un procedimiento que pasa por referencia ListaExpresiones, que selecciona el árbol correspondiente a ese lugar de la lista, libera su memoria dejándolo vacío y luego elimina el nodo de la lista, dejándola vacía.
Finalmente se emite un mensaje por pantalla para el usuario “hasta la próxima”
En el main se ejecuta una función llamada esExit() que deja cargado con el resultado TRUE a un booleano “es” en el main, ya que ListaStrings se vacía en cada do{}while del main mientras es valga FALSE. De esa manera se sale del loop y al final del main borra ListaStrings.

## 2. Diagrama de módulos



## 3. Pseudocódigo de la ejecución de cada comando de la aplicación

### 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 strcrear (String &s);` ← Módulo String

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

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

```
boolean streq (String s1, String s2) ← Módulo Strin

boolean transformarStringABoolean (String s) ← Módulo String

boolean esVacio (String s)  ← Módulo String

void eliminarBlancosPrincipio (String s, String &sb) ← Módulo String

void dividirString (String s, String &primero, String &resto)← Módulo String

boolean transformarStringABoolean (String s) ← Módulo String

void Crear (ListaStrings &L) ← Módulo Lista Strings

boolean Vacia(ListaStrings L) ← Módulo Lista Strings

String Primero(ListaStrings L ← Módulo Lista Strings

void Resto(ListaStrings &L) ← Módulo Lista Strings

void InsBackIter(String s, ListaStrings &L) ← Módulo Lista Strings

int largoListaStrings (ListaStrings L) ← Módulo Lista Strings

boolean PerteneceIter (String s, ListaStrings L)← Módulo Lista Strings

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

String darPalabraDeLista (String s, ListaStrings L) ← Módulo Lista Strings

int largoListaExpresiones (ListaExpresiones LE); ← Módulo Lista Expresiones

int largoListaExpresiones (ListaExpresiones L) ← Módulo ListaExpresiones

String darPalabraporPosicion (int posicion, ListaStrings L) ← Módulo ListaExpresiones
```

```
void atomic (ListaStrings L, ListaExpresiones &LE, Expresion &e, ArbolExpre &ar)
```
← Módulo ListaExpresiones

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

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

```
void insertarValorNodoArbol(ValorNodo vn, ArbolExpre &a);
```
← Módulo ArbolExpre

```
void asignarNumeroExpresion (Expresion &e, int num);
```
← Módulo Módulo Expresion

```
void asignarArbolExpresion (Expresion &e, ArbolExpre ar);
```
← Módulo Expresion

```
void insertarNodoEnlista (Expresion e, ListaExpresiones &LE);
```
← Módulo Lista de Expresiones

```
void desplegarPorNumero (ListaExpresiones LE, int numero);
```
← Módulo Lista de Expresiones

```
void error (errores codigo, int comando, ListaStrings L);
```
← Módulo Errores

**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
        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
            Si el total de palabras de la línea es 3 y la segunda palabra es “NOT”, entonces
                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 strcrear (String &s); ← Módulo String

void strdestruir(String &s); ← Módulo String

void print (String s); ← Módulo String

boolean streq (String s1, String s2) ← Módulo Strin

boolean transformarStringABoolean (String s) ← Módulo String

boolean esVacio (String s)  ← Módulo String

void eliminarBlancosPrincipio (String s, String &sb) ← Módulo String

void dividirString (String s, String &primero, String &resto)← Módulo String

boolean transformarStringABoolean (String s) ← Módulo String

boolean esNatural (String s)← Módulo String

int transformarANatural (String s) ← Módulo String

void Crear (ListaStrings &L) ← Módulo Lista Strings

boolean Vacia(ListaStrings L) ← Módulo Lista Strings

String Primero(ListaStrings L ← Módulo Lista Strings

void Resto(ListaStrings &L) ← Módulo Lista Strings

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

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

boolean PerteneceIter (String s, ListaStrings L)← Módulo Lista Strings

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

String darPalabraDeLista (String s, ListaStrings L) ← Módulo Lista Strings

int largoListaExpresiones (ListaExpresiones LE); ← Módulo Lista Expresiones

String darPalabraporPosicion (int posicion, ListaStrings L) ← Módulo ListaExpresiones

void compound(ListaStrings L, ListaExpresiones &LE, Expresion &e, ArbolExpre &ar) ← Módulo
ListaExpresiones

int PosicionListaString (String s, ListaStrings L); ← Módulo Lista Strings

boolean PerteneceAListaExpreConID (int Id, ListaExpresiones LE) ← Módulo ListaExpresiones

void crearArbol (ArbolExpre &a); ← Módulo ArbolExpre

void copiarArbolAOtro(ArbolExpre b, ArbolExpre &a) ← Módulo ArbolExpre

void insertarValorNodoArbol(ValorNodo vn, ArbolExpre &a)← Módulo ArbolExpre

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

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

void insertarOperadorNodo (char o, ValorNodo &valor) ← Módulo ValorNodo

void insertarParentesisNodo (char p, ValorNodo &valor) ← Módulo ValorNodo
```

```
void InsertarParentesisEnMinimo(char parentesis, ArbolExpre &a)
void InsertarParentesisEnMaximo(char parentesis, ArbolExpre &a)
void crearLista (ListaExpresiones &LE);
ArbolExpre seleccionarArbolExpre (Expresion e);
Expresion darExpresionConID (int Id, ListaExpresiones LE);
void asignarNumeroExpresion (Expresion &e, int num);
void asignarArbolExpresion (Expresion &e, ArbolExpre ar);
void insertarNodoEnlista (Expresion e, ListaExpresiones &LE);
void desplegarPorNumero (ListaExpresiones LE, int numero);
void error (errores codigo, int comando, ListaStrings L);
```

← Módulo ArbolExpre

← Módulo ArbolExpre

← Módulo Lista de Expresiones

← Módulo Expresion

←Módulo Lista de Expresiones

← Módulo Módulo Expresion

← Módulo Expresion

← Módulo Lista de Expresiones

← Módulo Lista de Expresiones

← Módulo Errores

**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 strcrear (String &s);
```
← Módulo String

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

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

```
boolean streq (String s1, String s2)
```
← Módulo String

```
boolean esVacio (String s)
void eliminarBlancosPrincipio (String s, String &sb)
void dividirString (String s, String &primero, String &resto)
void Crear (ListaStrings &L)
boolean Vacia(ListaStrings L)
String Primero(ListaStrings L
void Resto(ListaStrings &L)
void InsBackIter(String s, ListaStrings &L)
int largoListaStrings (ListaStrings L)
boolean PerteneceIter (String s, ListaStrings L)
void partirStrings (String s, ListaStrings &L)
String darPalabraporPosicion (int posicion, ListaStrings L)
int PosicionListaString (String s, ListaStrings L);
void show (ListaStrings L, ListaExpresiones LE, Expresion e, ArbolExpre ar)
void desplegarPorNumero (ListaExpresiones LE, int numero);
int seleccionarNumeroExpresion (Expresion e)
```

← Módulo String

← Módulo String

← Módulo String

← Módulo Lista Strings

← Módulo Lista Strings

← Módulo Lista Strings

← Módulo Lista Strings

← Módulo Lista Strings

← Módulo Lista Strings

← Módulo Lista Strings

← Módulo Lista Strings

← Módulo Lista Strings

← Módulo Lista Strings

← Módulo ListaExpresiones

← Módulo Lista Strings

← Módulo ListaExpresiones

← Módulo Lista de Expresiones

← Módulo Lista de Expresiones

```
void desplegarExpresion (Expresion e)
```
← Módulo Lista de Expresiones

```
void error (errores codigo, int comando, ListaStrings L);
```
← Módulo Errores

**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 índice que ingrese el usuario
                    Si el nodo es de tipo “VALOR”, entonces
                        retornar valor.
                    Sino
                        Si valor de nodo es tipo “OPERADOR”, entonces
                            Si  es “A”, entonces
                                Retornar evaluarExpresion(hder) && evaluarExpresion(hizq)
                            Sino
                                Si es “O”, entonces
                                    Retornar evaluarExpresion(hder) || evaluarExpresion(hizq)
                                Sino
                                    Si es ‘N’, entonces
                                        Retornar !evaluarExpresion(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 strcrear (String &s);
```
← Módulo String

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

```
boolean streq (String s1, String s2)
```
← Módulo Strin

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

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

```
void eliminarBlancosPrincipio (String s, String &sb)
```
← Módulo String

```
void dividirString (String s, String &primero, String &resto)
```
← Módulo String

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

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

```
boolean Vacia(ListaStrings L)
```
← Módulo Lista Strings

```
String Primero(ListaStrings L
```
← Módulo Lista Strings

```
void Resto(ListaStrings &L)
```
← Módulo Lista Strings

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

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

```
boolean PerteneceIter (String s, ListaStrings L)
```
← Módulo Lista Strings

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

```
String darPalabraporPosicion (int posicion, ListaStrings L)
```
← Módulo ListaExpresiones

```
void evaluate(ListaStrings L, ListaExpresiones LE, Expresion e, ArbolExpre ar)
```
← Módulo ListaExpresiones

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

```
boolean PerteneceAListaExpreConID (int Id, ListaExpresiones LE)
```
← Módulo ListaExpresiones

```
ArbolExpre seleccionarArbolExpre (Expresion e);
```
← Módulo Expresion

```
boolean evaluarExpresion(ArbolExpre a);
```
← Módulo ÁrbolExpre

```
TipoNodo darDiscriminante (ValorNodo valor)
```
← Módulo ValorNodo

```
char darOperadorArbol (ValorNodo valor)
```
← Módulo ValorNodo

```
boolean darBooleanoArbol (ValorNodo valor)
```
← Módulo ValorNodo

```
void error (errores codigo, int comando, ListaStrings L);
```
← Módulo Errores

**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, bajando a archivo cada nodo con su respectiva información
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, bajando a archivo cada nodo con su respectiva información
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 strcrear (String &s);
```
← Módulo String

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

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

```
boolean streq (String s1, String s2)
```
← Módulo Strin

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

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

```
void eliminarBlancosPrincipio (String s, String &sb)
```
← Módulo String

```
void dividirString (String s, String &primero, String &resto)
```
← Módulo String

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

```
boolean esNatural (String s)
int transformarANatural (String s)
void Crear (ListaStrings &L)
boolean Vacia(ListaStrings L)
String Primero(ListaStrings L
void Resto(ListaStrings &L)
void InsBackIter(String s, ListaStrings &L)
int largoListaStrings (ListaStrings L)
boolean PerteneceIter (String s, ListaStrings L)
void partirStrings (String s, ListaStrings &L)
String darPalabraporPosicion (int posicion, ListaStrings L)
void save (ListaStrings L, ListaExpresiones LE, Expresion e, ArbolExpre ar)
int PosicionListaString (String s, ListaStrings L);
boolean PerteneceAListaExpreConID (int Id, ListaExpresiones LE)
ArbolExpre seleccionarArbolExpre (Expresion e);
Expresion darExpresionConID (int Id, ListaExpresiones LE);
void dividirStringDeArchivo (String s, String &primero, String &resto)
```

← Módulo String

← Módulo String

← Módulo Lista Strings

← Módulo Lista Strings

← Módulo Lista Strings

← Módulo Lista Strings

← Módulo Lista Strings

← Módulo Lista Strings

← Módulo Lista Strings

← Módulo Lista Strings

← Módulo ListaExpresiones

← Módulo ListaExpresiones

← Módulo Lista Strings

← Módulo ListaExpresiones

← Módulo Expresion

←Módulo Lista de Expresiones

←Módulo Lista de Expresiones

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

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

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

```
boolean ExisteArchivo (String nombrearchivo)
```
← Módulo Archivo

```
void enumerarNodos(int &indice, ArbolExpre &a)
```
← Módulo ÁrbolExpre

```
void BajarArbolExpre (ArbolExpre a, FILE *f)
```
← Módulo Archivo

```
void bajarValorNodo (ValorNodo v, FILE *f)
```
← Módulo Archivo

```
void error (errores codigo, int comando, ListaStrings L);
```
← Módulo Errores

**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 x:”
                Recorrer el árbol recursivamente en órden y listar su contenido por pantalla.
            Fin
        Fin
    Fin
Fin

```
void scan (String &s);
void strcrear (String &s);
void strdestruir(String &s);
void print (String s);
boolean streq (String s1, String s2)
boolean transformarStringABoolean (String s)
boolean esVacio (String s)
void eliminarBlancosPrincipio (String s, String &sb)
void dividirString (String s, String &primero, String &resto)
boolean transformarStringABoolean (String s)
void Crear (ListaStrings &L)
boolean Vacia(ListaStrings L)
String Primero(ListaStrings L
void Resto(ListaStrings &L)
void InsBackIter(String s, ListaStrings &L)
int largoListaStrings (ListaStrings L)
boolean PerteneceIter (String s, ListaStrings L)
void partirStrings (String s, ListaStrings &L)
```

← Módulo String
← Módulo String
← Módulo String
← Módulo String
← Módulo Strin
← Módulo String
← Módulo String
← Módulo String
← Módulo String
← Módulo String
← Módulo Lista Strings
← Módulo Lista Strings
← Módulo Lista Strings
← Módulo Lista Strings
← Módulo Lista Strings
← Módulo Lista Strings
← Módulo Lista Strings
← Módulo Lista Strings

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

String darPalabraporPosicion (int posicion, ListaStrings L) ← Módulo ListaExpresiones

void load (ListaStrings L, ListaExpresiones &LE, Expresion &e, ArbolExpre &ar) ← Módulo ListaExpresiones

int PosicionListaString (String s, ListaStrings L); ← Módulo Lista Strings

void crearArbol (ArbolExpre &a); ← Módulo ArbolExpre

void dividirStringDeArchivo (String s, String &primero, String &resto)  ←Módulo Lista de Expresiones

boolean esAlfabetico (String s)← Módulo String

boolean tieneExtension (String s) ← Módulo String

boolean esNombreArchivo (String s) ← Módulo String

boolean ExisteArchivo (String nombrearchivo) ← Módulo Archivo

void LevantarArbolExpre (ArbolExpre &a, String nombrearchivo) ← Módulo Archivo

void levantarValorNodo (ValorNodo &v, FILE *f) ← Módulo Archivo

void insertarValorEnOrden(ValorNodo valor, ArbolExpre &a) ← Módulo ArbolExpre

TipoNodo darTipo (TipoNodo tipo) ← Módulo ValorNodo

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

ArbolExpre seleccionarArbolExpre (Expresion e); ← Módulo Expresion

void asignarNumeroExpresion (Expresion &e, int num); ← Módulo Módulo Expresion
```

```
void asignarArbolExpresion (Expresion &e, ArbolExpre ar);
```
← Módulo Expresion

```
int seleccionarNumeroExpresion (Expresion e)
```
← Módulo Expresion

```
void insertarNodoEnlista (Expresion e, ListaExpresiones &LE);
```
← Módulo Lista de Expresiones

```
void desplegarPorNumero (ListaExpresiones LE, int numero);
```
← Módulo Lista de Expresiones

```
void error (errores codigo, int comando, ListaStrings L);
```
← Módulo Errores

**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
Fin

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

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

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

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

```
boolean streq (String s1, String s2)
```
← Módulo String

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

```
void eliminarBlancosPrincipio (String s, String &sb)
```
← Módulo String

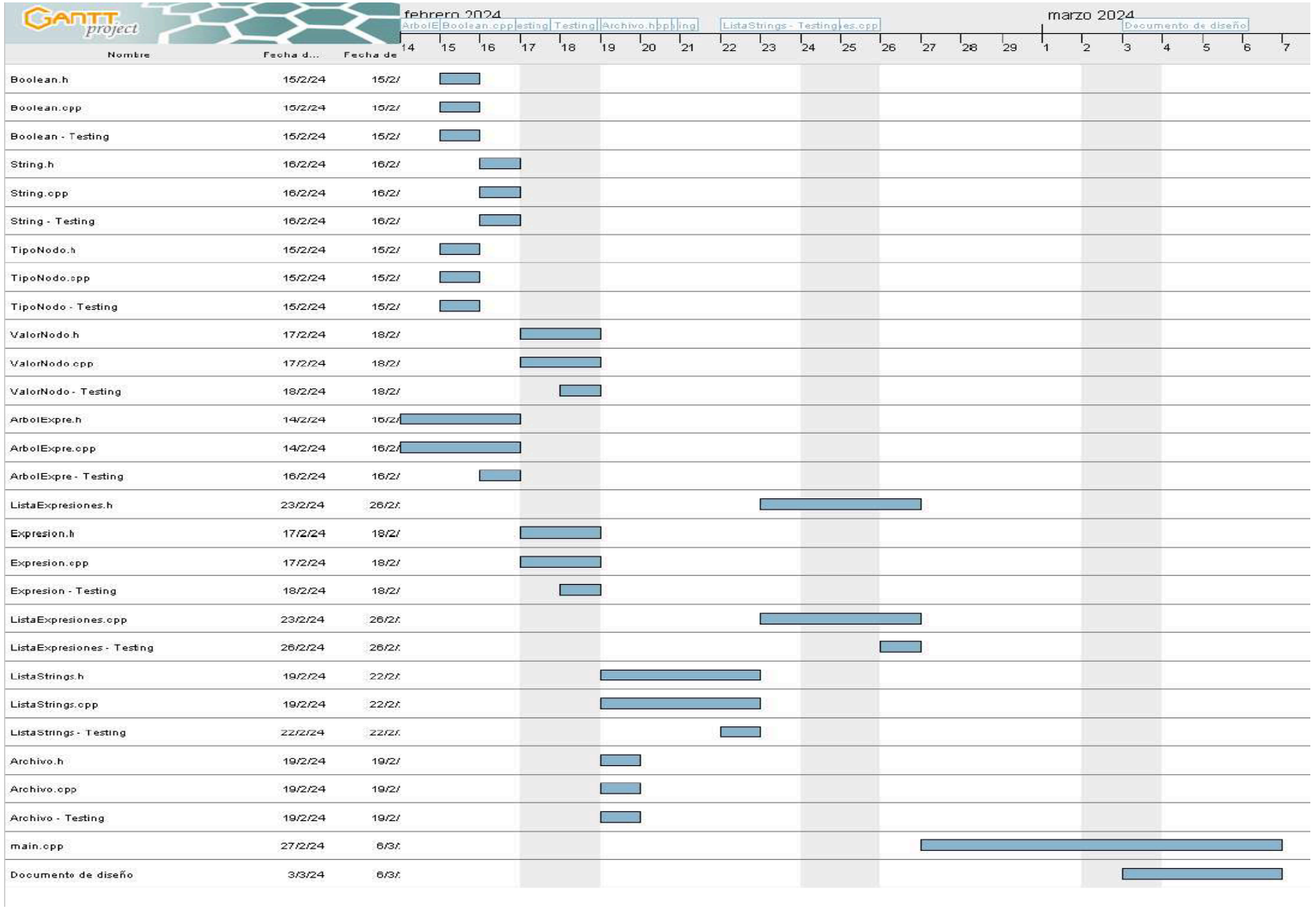
```
void dividirString (String s, String &primero, String &resto)
```
← Módulo String

```
void Crear (ListaStrings &L)
```
← Módulo Lista Strings

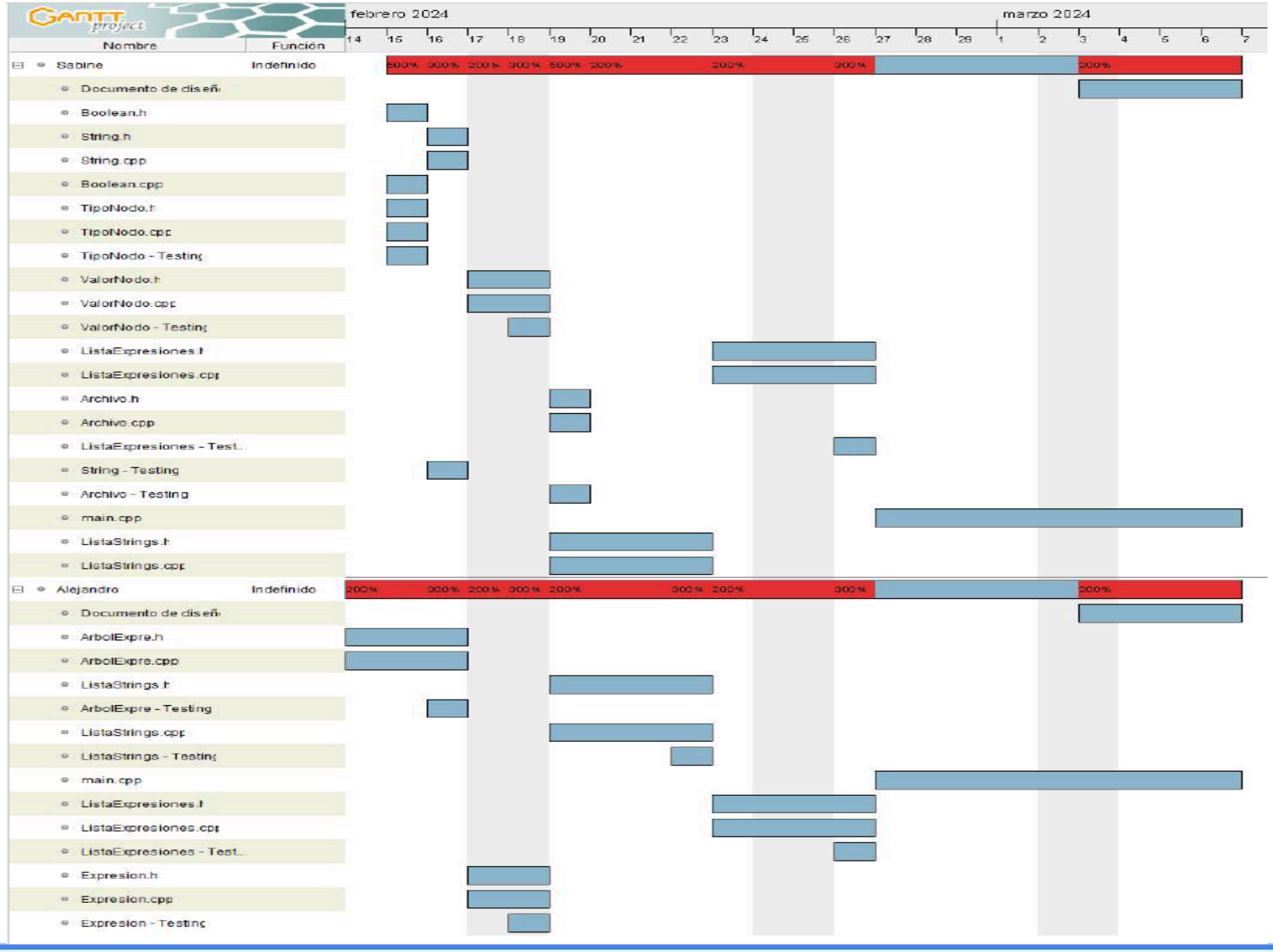
```
boolean Vacia(ListaStrings L) ← Módulo Lista Strings

String Primero(ListaStrings L ← Módulo Lista Strings

void Resto(ListaStrings &L) ← Módulo Lista Strings

void InsBackIter(String s, ListaStrings &L) ← Módulo Lista Strings

int largoListaStrings (ListaStrings L) ← Módulo Lista Strings

boolean PerteneceIter (String s, ListaStrings L)← Módulo Lista Strings

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

int PosicionListaString (String s, ListaStrings L); ← Módulo Lista Strings

void exit(ListaStrings L, ListaExpresiones LE, Expresion e, ArbolExpre ar) ← Módulo ListaExpresiones

void liberarMemoriaListaE (ListaExpresiones &LE) ← Módulo Lista de Expresiones

ArbolExpre seleccionarArbolExpre (Expresion e); ← Módulo Expresion

void liberarMemoriaArbol(ArbolExpre &a) ← Módulo rbolExpre

void error (errores codigo, int comando, ListaStrings L); ← Módulo Errores
```

## 4. Cronograma de implementación y testing

Vista general:

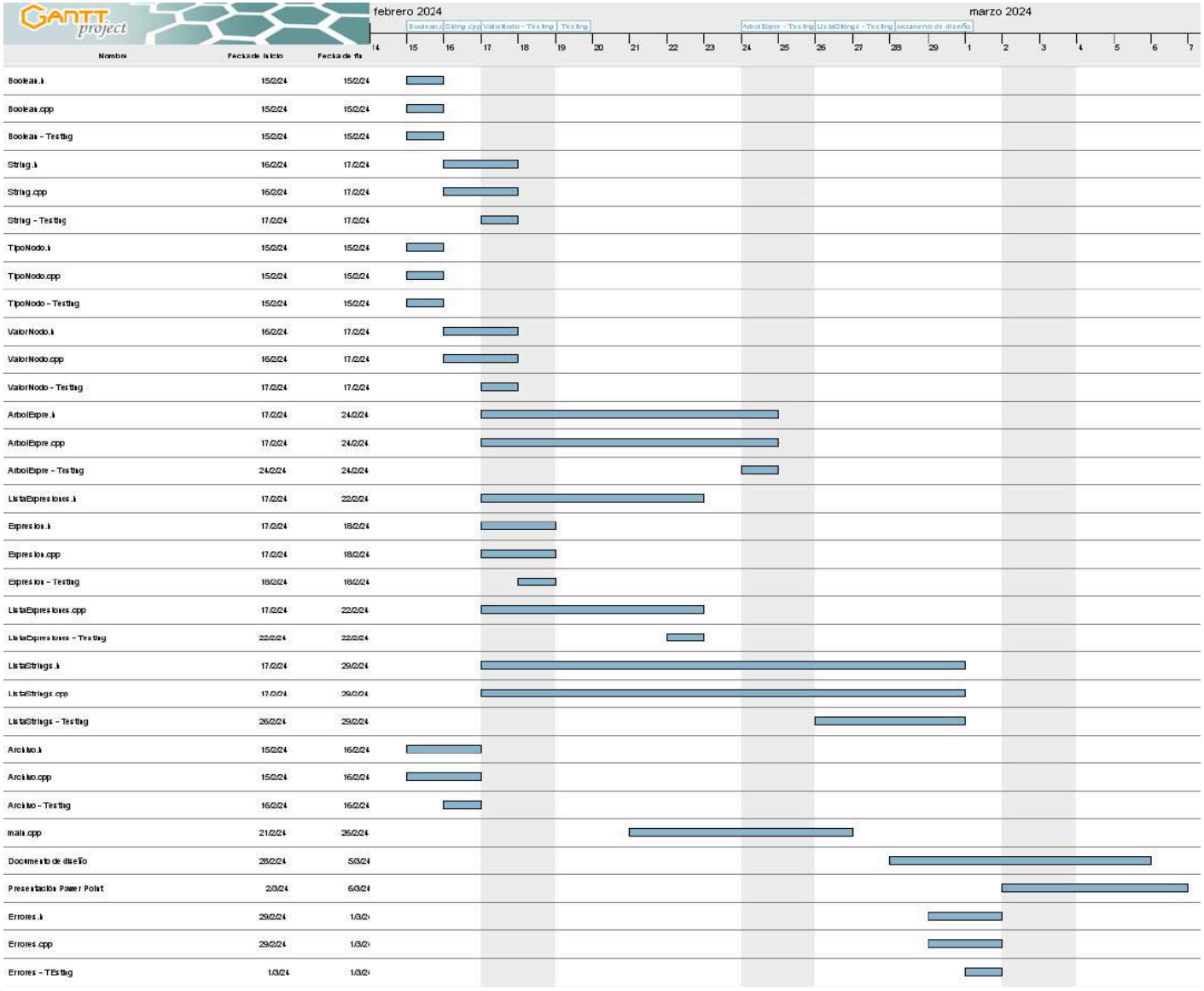


Vista por persona:

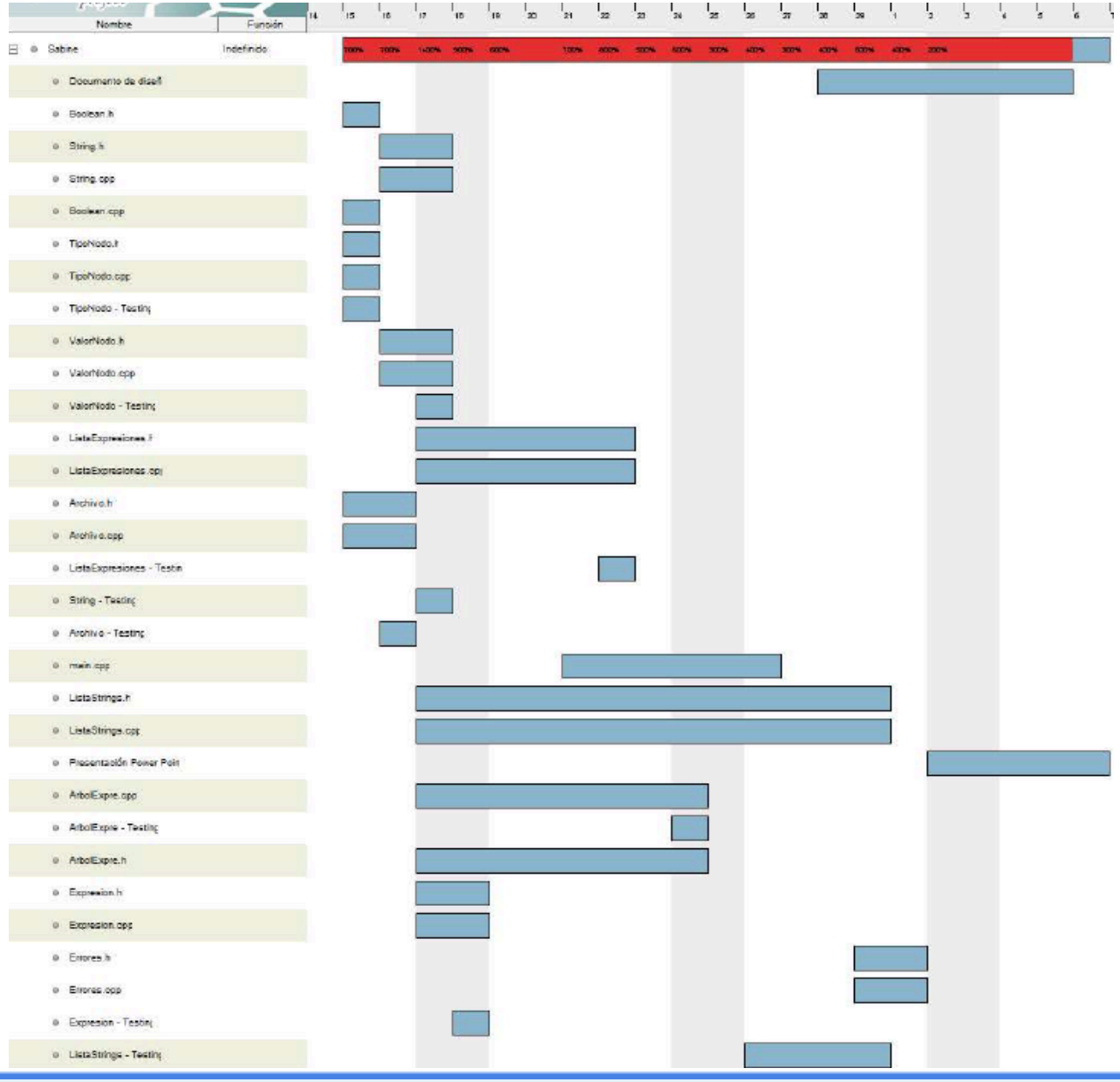


## 5. Cronograma luego de la implementación

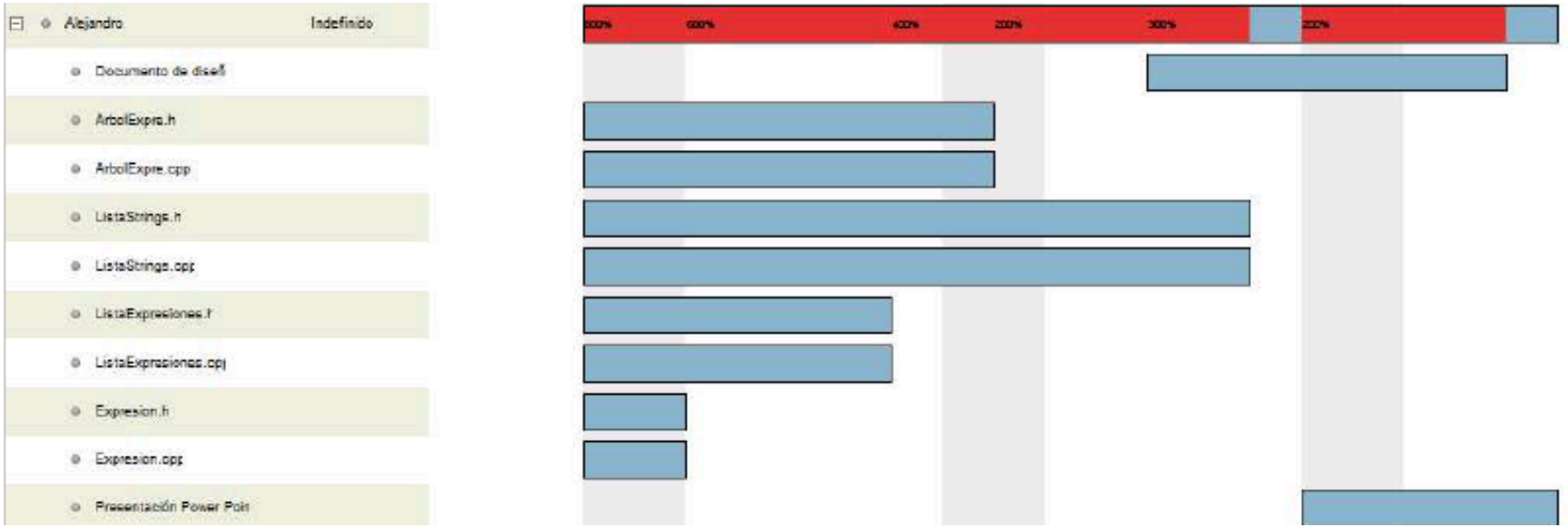
Vista general:



Vista por persona:
(Sabine)



(Alejandro)


Alejandro
Indefinido
Documento de diseñ
ArbolExpre.h
ArbolExpre.cpp
ListaStrings.h
ListaStrings.cpp
ListaExpresiones.h
ListaExpresiones.cpp
Expresion.h
Expresion.cpp
Presentación Power Poin

## 6. Proceso de creación del programa, problemas enfrentados y soluciones encontradas

Se repartió el trabajo a realizar según cronograma por los diferentes módulos.

**Booleano**
El módulo Booleano no dio mayor trabajo, ya que se podía utilizar el mismo módulo que se usó en clase.

**String**
El módulo String resultó más complejo de lo estimado por la necesidad de varias operaciones de conversión (String a char, String a boolean, verificación si un caracter es natural o no, verificar si el nombre de un archivo es alfabético, verificar si la extensión es .dat) además de eliminar los blancos al principio de un String y dividir el String en dos. Si bien pudimos resolver relativamente rápido todo, se gastó más tiempo de lo estipulado en este módulo. Lo que generó mayores problemas fue la verificación si un String era un número natural, para lo cual se utilizó una función de la biblioteca Stlib.h (atoi()), que nos ayudó a resolver este problema.

**TipoNodo**
El módulo TipoNodo fue terminado en el tiempo estipulado, tratándose de un simple enumerado y no dio considerable trabajo.

**ValorNodo**
El módulo ValorNodo si bien se terminó en el tiempo estipulado (se empezó a trabajar antes de lo previsto y por eso se terminó antes), dio más trabajo de lo estimado, ya que hacían falta varias operaciones adicionales que no habíamos contemplado, como discriminador de los diferentes tipos de operador (AND, OR, NOT), los diferentes tipos de paréntesis. Además nos dimos cuenta que en este módulo tenían que ir las asignaciones que luego se iban a llamar por cada nodo del árbol. Otra cosa notada más adelante fue la necesidad de tener un procedimiento de asignar un índice a cada nodo para luego poder guardarlo en archivo con ese índice.
El tener este módulo pronto relativamente temprano alivió mucho el trabajo para las operaciones de ArbolExpre.

**Archivo**
Se optó por poner las operaciones relacionados archivo en un módulo separado llamado Archivo, el mismo se realizó sin mayores problemas, una vez que nos quedó claro el proceso:

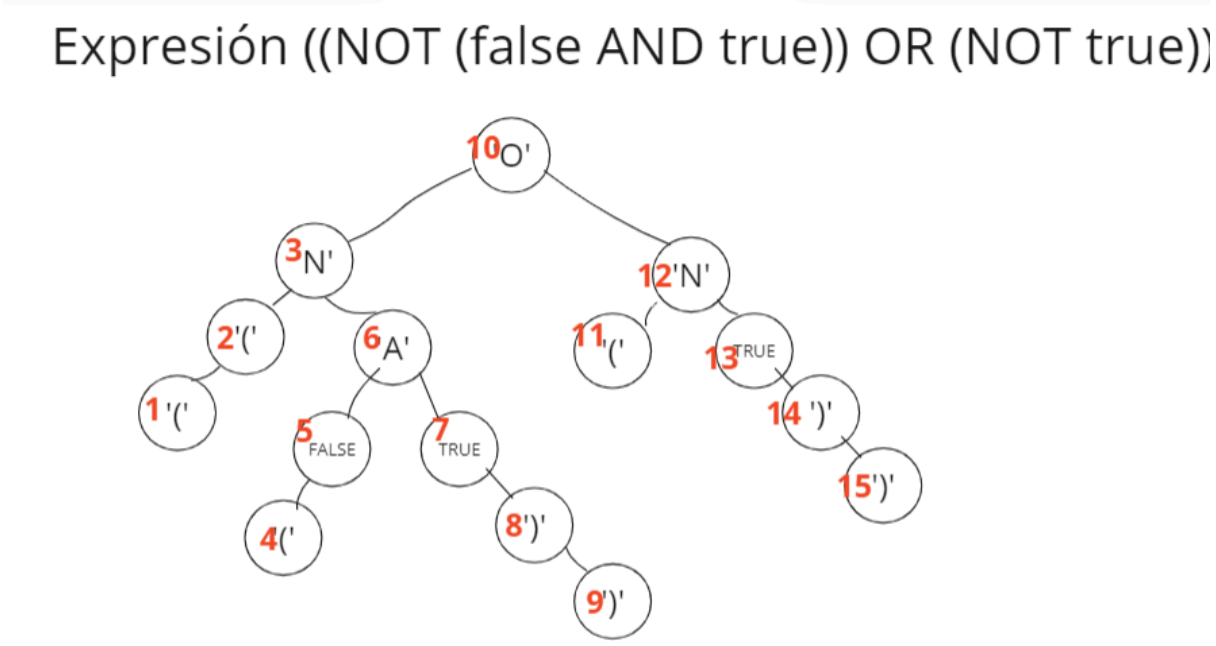
1) Tener los índices de los nodos asignados para bajarlos a archivo en el “load”
2) Teníamos que bajar esos índices al archivo para que queden guardados junto a los demás datos de cada nodo y que la bajada tenía que ser en preorden.
3) Al subir el árbol de archivo a memoria, lo teníamos que subir en órden según los índices previamente guardados en el archivo

Para realizar estos procedimientos, era de mucha utilidad ya tener el módulo ValorNodo pronto, ya que la bajada o subida del árbol utiliza procedimientos repetinos de ese módulo.
Nos dimos cuenta que una vez guardado correctamente en archivo, el proceso de subir el árbol a memoria era relativamente sencillo y pudimos implementar el procedimiento en cuestión (parte de ArbolExpre) relativamente rápido y mucho antes de los métodos para “armar” el árbol si no se encuentra previamente guardado en archivo.

**Proceso de bajar árbol a archivo:**

Enumerar nodos (procedimiento implementado en ArbolExpre que utiliza el procedimiento de asignar índice de ValorNodo) **en orden**.

Luego se baja cada nodo del árbol a archivo en **preorden**, llamando al procedimiento de bajarValorNodo en cada nodo, en el ejemplo quedando así guardado en el archivo:




**10** ‘O’
**3** ‘N’
**2** ‘(‘
**1** ‘(‘
**6** ‘A’
**5** FALSE
**4** ‘(‘
**7** TRUE
**8** ‘)’
**9** ‘)’
**12** ‘N’
**11** ‘(‘
**13** TRUE
**14** ‘)’
**15** ‘)’

El índice se guarda como un entero y el contenido del estructurado como un char (paréntesis, operador) o como un boolean (valor), con un switch (por ser necesario indicar el tipo de dato a guardar en archivo para bajarlo).
Cuando se levanta a memoria el contenido del archivo, se levanta **en orden** según los índices (numeritos rojos) asignados como si fuera un árbol binario de búsqueda. Para levantar cada nodo, se llama al procedimiento de levantarValorNodo() una vez al principio, luego mientras no se haya llegado al final del archivo, se llama al procedimiento insertarValorEnOrden() de ArbolExpre que en el caso del árbol vacío directamente inserta el contenido. Si no, compara el índice del nodo levantado con el contenido del nodo del árbol y según si es mayor o menor, lo inserta a la derecha o a la izquierda.
Se sigue llamando levantarValorNodo() y repitiendo el proceso hasta que se llegue al final del archivo.

**ArbolExpre**

El módulo de ArbolExpre estaba en principio asignado a uno de los dos integrantes solamente, pero terminamos trabajando en conjunto por ser más extenso y dificultoso de lo que se había estipulado. Nos dimos cuenta que teníamos que implementar operaciones de inserción diferentes según si el índice del nodo se encontraba cargado (operación que nos causó menos problemas, ya que se trata como un árbol binario de búsqueda y según su índice se inserta el contenido a la izquierda o a la derecha del nodo existente). Y por otro lado si solamente se va a insertar un valor (TRUE o FALSE, nuestro boolean de tipo enum) sin paréntesis (en el comando “atomic”) o si teníamos que “juntar” dos árboles existentes unidas por un operador en una expresión compuesta (en el comando “compound”).
Dicho comando fue el que en general nos costó más trabajo y tiempo y nos resultó más dificultoso de resolver. Finalmente entendimos que en ese comando había que crear primero un nuevo árbol vacío, insertarle el operador en cuestión (mediante el procedimiento en ValorNodo que lo discrimina) y dos hijos en principio vacíos. Luego tratar cada subarbol que se va a unir a través de un operador (AND, OR, NOT) de forma “genérica”, copiándolo (no importando el contenido de ValorNodo en ese momento) con un procedimiento recursivo en el caso del NOT al hijo derecho del nuevo árbol, o en el caso del AND u OR al hijo izquierdo la parte izquierda al operador y en el hijo derecho la parte derecha al operador. Finalmente se insertan los paréntesis de apertura (en el hijo más izquierdo del nuevo árbol) y de cierre (en el hijo más derecho del nuevo árbol). Para la inserción de los paréntesis, utilizamos otro procedimiento dentro del mismo módulo, que se asegura de que siempre van a estar en los extremos del árbol.
Otra procedimiento que nos costó fue el de enumerar los nodos para la inserción en el archivo, ya que el mismo tiene que asignar correctamente los índices de los nodos en orden incrementándolos en cada nodo que va avanzando. Lo resolvimos seteando afuera del procedimiento el índice como 0, luego recursivamente recorrer el subárbol izquierdo, en la raíz sumar uno al índice, llamar al procedimiento de asignarIndiceNodo() de ValorNodo sobre la raíz, y recorrer recursivamente el subárbol derecho.
Otra dificultad tuvimos con resolver el procedimiento de evaluarExpresion(), ya que nos devolvía el booleano de C (bool) en vez de nuestro Booleano boolean. Lo resolvimos retornando específicamente nuestro booleano (enumerado) en los pasos recursivos de la función, según su resultado.

**Expresion**

El módulo Expresion si bien solo es un estructurado, nos tuvo “trancados” por unos días. Inicialmente también solo asignado a uno de los dos integrantes, terminamos trabajando en conjunto en él para cumplir con el plazo. Lo que más problema nos dio, si bien tiene solución relativamente sencilla, fue implementar el procedimiento asignarArbolExpresion() que asigna el árbol al miembro de la Expresión. Este procedimiento fue el paso que nos faltaba para poder implementar varias otras operaciones.

También nos dio problemas implementar el desplegarExpresion() porque le estábamos pasando el árbol en vez de la selectora del árbol. Por no poder lidiar con estos problemas, nos atrasamos con otros módulos más de lo previsto.

**ListaStrings**

Con ListaStrings empezamos a trabajar antes de lo planificado pero pasamos más tiempo de lo estipulado por ser mucho más complejo de lo que pensamos (a pesar de ya ser uno de los módulos que más tiempo asignamos desde el comienzo). Luego de hacer el cronograma, nos pedían tener cierta parte de este módulo ya resuelta (que requería de varios procedimientos del módulo String) que habíamos previsto hacer recién más adelante, pero fue bueno empezar a trabajar antes en este módulo. Pudimos resolver relativamente rápido lo que nos pidieron al principio (operaciones para devolver el primero y el resto de la Lista, saber si es vacía o no, insertar un nuevo String al final de la lista). Con esos procedimientos armamos el partirStrings() que lo necesitamos para el parseo.

Este módulo fue el que más tiempo nos llevó, debido a que optamos no tener un main gigante que nos parecía casi imposible de manejar con incontables if…else, sino dividir por comandos que va ingresando el usuario (atomic, compound, show, avaluate, save, load, exit), una vez que se detecta un comando (o sea, el comando pertenece a ListaStrings), se llama el procedimiento acorde. Pensamos que de esta manera quedó mucho más ordenado todo, pero hizo mucho más grande y trabajoso el módulo ListaStrings de lo anteriormente previsto.

Hicimos un procedimiento por cada comando con la mayoría de los chequeos (parseo) adentro de cada procedimiento. Todos ellos forman parte de ListaStrings. En cada procedimiento, luego de pasar los chequeos correspondientes, se llaman las funciones y procedimientos necesarios para ejecutar el comando. Nos orientamos casi al pie de la letra en el pseudocódigo para realizar los chequeos y resultó bastante útil. Si bien el parser fue trabajoso en armar, no nos resultó muy difícil luego de saber cómo realizar la parte de separar las palabras.

Una operación que nos dio bastantes problemas fue la de devolver una palabra por posición dada en ListaStrings, intentamos resolver esto utilizando otros procedimientos como la de devolver una palabra de la lista dada la palabra, pero no nos funcionó. Terminamos implementando el procedimiento de darPalabraporPosicion() con un procedimiento mucho más simple (aumentando una variable contadora en cada nodo que pasaba mientras la misma no era igual a la posición buscada, devolviendo el String solamente cuando ambas variables eran iguales). Con este procedimiento funcionando y varios otros que armamos (la mencionada obtener la palabra dada de la lista, devolver largo de la lista y un procedimiento que devuelve un booleano según si un String dado pertenece a la lista o no, pudimos armar los diferentes comandos (atomic, compound, show, evaluate, save, load, exit) que se llaman desde el main siempre que esa palabra pertenece a ListaStrings.

Otro de los problemas que tuvimos con este módulo, fue cuando quisimos seleccionar el árbol dentro de la Expresión, dentro de ListaExpresiones (en el caso de querer bajar a archivo el contenido de una posición dada de ListaExpresiones o en el caso de borrar la memoria del árbol dentro de una posición dada de Listaexpresiones). Lo solucionamos asignando primero el árbol a una variable y utilizamos la variable en vez de la selectora.

**ListaExpresiones**

Para ListaExpresiones habíamos plainificado un poco menos tiempo de lo que nos llevó al final, terminamos este módulo 5 días antes de lo previsto por haber empezado antes de trabajar en él, debido a que la gran mayoría de procedimientos requeridos ya se había abordado en clase. Implementamos solamente una solectora por ID que determinaba si un número entero (el ID) pertenece a La Lista y un procedimiento parecido que devolvía dicho ID.

Las operaciones que había que realizar sobre la lista, con Expresion y ArbolExpre resulto, eran fáciles de implementar. El programa no requería que se guarde o despliegue más de un ID ni se acceda más que uno a la vez, por lo tanto no los procedimientos en este módulo son bastante acotados.

El mayor problema que tuvimos con este módulo, ya fue tratado en Expresion.h (porque tiene el origen ahí). También el problema que tuvimos con ArbolExpre (para borrar el árbol dentro del estructurado Expresion dentro de la ListaExpre) ya se explicó más arriba.

**main**

El main nos llevó mucho menos tiempo de lo previsto y ya lo teníamos pronto mucho antes de lo estipulo. Esto es debido a que solamente se llaman en el main los diferentes procedimientos (atomic, compound, evaluate, show, save, load, exit). La mayor parte de cheque (parser) se realiza en cada procedimiento an ListaStrings.

En el main tuvimos un pequeño problema porque al borrar la ListaStrings cada vez que se ingrese otro línea, no se salía del do {}while loop. Esto lo arreglamos armando una función “esExit()” que pertenece a ListaStrings y devuelve “TRUE” si se pasan los chequeos pertinentes al exit. Para salir del do{}while se tiene que cumplir la condición de “es” (equivalente a “esExit valiendo “TRUE”, es decir el usuario ingresó el comando exit de forma válida). Si se cumple la condición, se sale del do{}while. Además solo se hace el borrado de ListaStrings por cada línea (cada iteración por el loop) si esExit vale “FALSE”, es decir el usuario no ingresó exit de forma válida. En el caso que sí lo hace, se borra ListaStrings recién después de salir del do{}while.

El borrado de ListaStrings no se está haciendo dentro del comando “exit”, para dejarla disponible para la evaluacion del do{}while.

**Errores**

Se decidió implementar el módulo **Errores** por sugerencia del docente de clase de consultas, para que el módulo ListaStrings que ya está bastante extenso, quede algo menos cargado. Para esto, en vez de printfs con los diferentes mensajes de error, se llama un procedimiento error() con el error en cuestión (que es un tipo enumerado que se asigna en cada caso de error al error correspondiente) y un entero que representa el comando en el cual se encuentra el error (ejemplo: atomic - 1, compound - 2). Además se pasa la ListaStrings para poder desplegar el mensaje de error correcto en algunos casos que el mensaje debe ser diferente según el contenido de ListaStrings, pero el comando es el mismo.

No habíamos previsto realizar este módulo, solamente lo implementamos porque nos encontramos bastante por delante del cronograma, con gran parte del programa funcionando.

## 7. Balance de trabajo y conclusiones del taller en general

En general fue una experiencia positiva que nos ayudó un montón en varios aspectos. Fuimos adquiriendo conocimientos sobre cómo realizar el parseado sobre un String y luego guardar los Strings en una lista, cosa que nunca habíamos hecho, y estimamos que sería más difícil de lo que resultó.

También aprendimos mucho sobre cómo construir un árbol binario que no es de búsqueda, ya que con los comandos “atomic” y “compound” vamos construyendo un árbol sin asignarle ningún índice. Nos sorprendió gratamente poder lidiar con ese problema relativamente rápido, pudiendo terminar esa parte antes de lo previsto.

Nos sirvió la ayuda brindada para el parser, que fue uno de los problemas que estimamos al principio más difícil de resolver, pero al final con esa guía lo pudimos resolver mucho más rápido de lo pensado.

En general si bien estuvimos estancados en algunos puntos, se trabajó mucho y dio gusto resolver los problemas planteados y que luego de muchas horas invertidas, funcione como debería. Nos resultó muy gratificante y enriquecedora la experiencia del armado del programa, si bien nos llevó mucho tiempo y trabajo.

Uno de los puntos positivos fue que nos obligó a planificar y llevar a cabo un cronograma de implementaciones, de esta forma teniendo cierto control sobre el avance del proyecto, saber si estábamos bien o atrasados y dividir el trabajo de antemano. Creemos que de esa forma se trabaja de una manera mucho más ordenada y controlada (si bien nos enfrentamos con algunos problemas y por eso tuvimos que redistribuir el trabajo un poco).

Algo negativo que nos pasó bastante al principio, fue que uno de los integrantes del grupo (originalmente fuimos 3), se bajó por problemas personales, quedando solo nosotros dos, lo cual fue negativo en general, ya que la carga de trabajo para cada uno de los restantes integrantes fue mayor que con tres y habíamos ya empezado a planificar la distribución de tareas en principio basado en 3 personas, teniendo que modificarla.

Lo que nos costó más y nos resultó algo difícil al principio (también no ayudó que en ese momento se nos bajó el tercer integrante), fue el proceso de pensar el pseudocódigo requerido para los diferentes comandos (sobre todo el compound), poder despedazarlo en partes y entender cómo podría llegar a funcionar. Nos fue brindada bastante información respecto a los tipos de datos a utilizar, pero tuvimos que consultar en varias ocasiones sobre el pseudocódigo de los comandos porque a pesar de reunirnos varios días y pensar en conjunto y por separado, nos encontramos sin ideas (o desarrollamos ideas que luego tuvimos que descartar porque no funcionaban de manera relativamente sencilla). En este aspecto fue negativo contar con muy pocas clases de apoyo y en general poco tiempo antes de la primera entrega.

Otro de las dificultades con las que nos encontramos fue que el editor de código a utilizar (Codeblocks) es muy dificultoso para instalarlo en una Mac (uno de los integrantes no contaba con computadora con sistema operativo Windows) y el editor de código que él utiliza (Visual Studio Code) no es compatible con Codeblocks y por ende le fue imposible compilar y ejecutar el programa, haciendo imposible hacer ciertas tareas. Uno de los requerimientos de la entrega final es entregar el programa en Codeblocks, creemos que sería bueno utilizar un programa compatible con otros sistemas operativos y/o editores de código, que no sean Windows, para evitar este tipo de problemas. Por esta razón cierto trabajo tenía que ser realizado siempre por la integrante del grupo que tenía Codeblocks (por tener Windows y por ende poder utilizar Codeblocks sin problemas) y no se pudo trabajar a la par como nos hubiera gustado y planificamos (la idea fue trabajar a la par en GITHub). Buscamos solucionar el tema, pero lamentablemente no le encontramos solución para hacer funcionar Codeblocks en Mac.
Por eso la parte de testeo tuvo que realizar enteramente la integrante del equipo con Codeblocks por las mismas dificultades explicadas anteriormente.

A pesar de las dificultades enfrentadas, concluimos positivamente, aprovechando todas las tutorías y las consultas de implementación de los miércoles para despejar dudas, y poder avanzar cuando estábamos trancados con la implementación de alguna funcionalidad puntual del programa. Si bien nos llevó un tiempo considerable, se llegó al resultado deseado, el proceso para llegar fue muy interesante y aprendimos mucho.