Trabajo Práctico N°2

Licenciatura en Ciencias de la Computación

Estructuras de datos y algoritmos I

Cátedra: Federico Severino Guimpel, Mauro Lucci, Martín Ceresa, Emilio López, Valentina Bini

Implementación de árboles de intervalos con intérprete

Lucas Bachur, Tomás Scalbi

2020

Índice

1. Introducción

1.1. Motivación del trabajo

Los objetivos del trabajo se pueden dividir en 2, el primero es desarrollar una implementación para *árboles de intervalos* valiéndose de **árboles AVL**, que soporte las funciones de insertar, elimnar e intersecar. Y luego la implementación de un intérprete para manipular este tipo de árboles desde la consola.

1.2. Breve descripción

Nuestro programa al que llamamos *interprete* toma comandos ingresados a través de la entrada estándar. Si el comando es válido realiza la acción correspondiente, en caso contrario muestra un mensaje de error. Los comandos válidos son:

- i [a,b]: inserta el intervalo [a, b] en el árbol.
- e [a,b]: elimina el intervalo [a, b] del árbol.
- ? [a,b]: intersecta el intervalo [a, b] con los intervalos del árbol. Imprime "No"si no hay intersección o imprime "Si, [x,y]"si la hay, donde [x, y] es un intervalo del árbol que intersecta a [a, b].
- dfs: imprime los intervalos del árbol con recorrido primero en profundidad.
- bfs: imprime los intervalos del árbol con recorrido primero a lo ancho.
- salir: destruye el árbol y termina el programa.

Ejecución Para ejecutar el programa, se debe introducir **make** en la consola, y luego ya se puede ejectuar **./interprete.out**. Hicimos el make con el objetivo de que en la carpeta principal solo aparezca el .out, y también incluimos los comandos cleanLin y cleanWin para llevar la carpeta al estado de antes de compilar.

1.3. Ejemplo de uso

```
C:\Users\Lucas\Documents\LCC\EyAD\EyDA-TP2>.\interprete.out
Buenas, bienvenido al interprete!
i [2,5]
dfs
 [2, 5]
i [2, 7]
dfs
 [2, 5] [2, 7]
i [-1e-4,1e4]
dfs
 [2, 5] [-0.0001, 10000] [2, 7]
bfs
 [2, 5] [-0.0001, 10000] [2, 7]
? [10,110]
Si, [-0.0001, 10000]
? [-20, -5]
No.
? [0,1e4]
Si, [2, 5]
e [-0.0001, 10000]
dfs
 [2, 5] [2, 7]
bfs
 [2, 5] [2, 7]
salir
Gracias por usar el interprete!
```

2. Carpeta de entrega

En la carpeta donde se entrega este trabajo, hay una sola carpeta que contiene varias dentro, y un makefile. Si bien podriamos haber dispuesto de las carpetas de forma diferente, esta nos pareció la más simple puesto que para incluir los distintos .h a lo largo de la implementación se utilizan los projects paths que son abstraibles con respecto a donde esté el proyecto.

Carpetas Como se puede ver en la imagen, en cada carpeta se encuentran un ".h" y ".c". El pimero es el archivo de cabecera con los prototipos y definiciones que encontramos necesarios para cada implementación. Y el segundo son las implementaciones de estas funciones declaradas en el ".h"
A su vez cada carpeta tiene en su interior, las carpetas con las implementaciones que necesita para su propia implementación, es decir por ejemplo, ITree/ contiene a Cola/ a Intervalos/ Pila/, a su propia implementacion, y el makefile que la compila.

makefile Antes de utilizar este método, tuvimos dificultades queriendo lograr que todos los ".o" nos queden en las carpetas donde se compilaban. Lo habíamos logrado en linux con un makefile único, pero no pudimos hacerlo funcionar en Windows. Según se nos recomendo en consulta, utilizamos un archivo makefile por cada carpeta que se encarga de compilar todos los archivos que allí se encuentren. Luego el makefile principal que esta a la misma altura que la carpeta "Interprete" se encarga de llamar a todos los demás.

3. Pilas y Colas

La inclusión de las implementaciones de Pilas y Colas en este trabajo se debió a la necesidad de utilizar estas estrucutras en los recorridos de DFS y BFS para los árboles de intervalo.

3.1. Colas

Implementación: Al principio nos costó decidir a que llamariamos comienzo y a que final. No estabamos seguros si los nodos debían apuntar a su "anterior" o a su "siguiente". Pero terminamos razonandolo según el ejemplo de una cola para un cajero. El comienzo de la cola es en el cajero, y la persona(nodo) que está enfrente, apunta hacia la persona que tiene detrás. El fin de la cola es donde se agregan las personas(nodos). Cada persona(nodo) tiene la información sobre quién está detrás, por lo tanto cuando se agrega una persona(nodo), el final actual de la cola(persona) debe ponerse al tanto que el lugar de atras dejo de estar vacio, y ahora se encuentra esta nueva persona(nodo).

3.2. Pilas

Implementación: La implementacion de pilas fue muy sencilla aunque también resultó un poco anti-intuitivo al comienzo la forma de desapilar y quitar elementos de la misma. Aunque la estructura base de la pila y cola que es:

```
1  typedef struct _GNodo {
2  void *dato;
3  struct _GNodo *next;
4  } GNodo;
5
```

Como tenían el mismo nombre, optamos por simplemente modificar el nombre de la estructura en el archivo de cabezera de la implementacion de pilas.

4. Intervalos

Implementación: Decidimos crear una implementacion de intervarlos de números reales que se pueda abstraer de los árboles. En su archivo.h explicamos bien su definición:

```
1  /**
2   * Representamos intervalos cerrados de numeros reales a traves de una
3   * estructura compuesta por dos doubles donde el primero representa el extremo
4   * izquierdo, y el segundo el extremo derecho del intervalo.
5   * Decidimos que el tipo de dato al que llamamos Intervalo, no sea un apuntador
6   * a la estructura _Intervalo puesto que este a diferencia de las demas tipos de
7   * datos declarados como GNodo, no se autoreferencian.
8   */
9  typedef struct _Intervalo {
10   double extIzq;
11   double extDer;
12 } Intervalo;
13
```

Funciones Definimos las funciones que consideramos necesarias en intervalos.h:

- **intervalo_crear**
- **intervalo_comparacion**, utilizada como *brújula* para buscar nodos en la eliminación, inserción.
- **intervalo_imprimir**, utilizada en el intérprete.
- **intervalo_interseccion** , utilizada para determinar si dos intervalos se intersecan.
- **intervalo_validar** , utilizada para validar si los números de un Intervalo son correctos, es decir, si están realmente representando un intervalo cerrado de números reales.

5. ITree

5.1. Estructura

Comenzamos el trabajo implementando AVLTrees solo de enteros, para introducirnos mejor en el trabajo con árboles. Luego la transición fue sencilla, e implementamos la función intersecar por último.

Decisiones Con respecto a la estructura de los arboles que empleamos, decidimos añadir *2 nuevos campos*, uno de **altura de árbol** que es un entero que almacena la altura del árbol, y el otro **máximo extremo derecho del árbol** que es un double que almacena el máximo extremo derecho del árbol.
Además decidimos implementar los árboles AVL para el caso particular de este trabajo, es decir con intervalos de doubles y no con apuntadores a vacío.
Nuestra estructura de árboles de intervalos termino así:

```
/**
 * Estructura que representa cada nodo (o subarbol) de un arbol de intervalos.
 * Cada nodo tiene:
 *  - un intervalo
 *  - un double que representa el maximo extremo derecho de ese subarbol
 *  - un int que representa la altura de ese subarbol
 *  - 2 apuntadores a 2 nodos que representan los hijos del subarbol actual
 */
typedef struct _ITreeNodo {
  Intervalo intervalo;
  double maxExtDer;
  int altura;
  struct _ITreeNodo *left;
  struct _ITreeNodo *right;
} ITreeNodo;

/**
 * Estructura de ITree.
 * El ITree (o arbol de intervalos) es un tipo de AVLTree que trabaja
 * especificamente con intervalos.
 */
typedef ITreeNodo *ITree;

```

Tomamos la decision de añadir el máximo extremo derecho segun se recomendó en el enunciado del trabajo.

Altura como parte de la estructura Y con respecto a la inclusión de la altura, lo hicimos para mejorar la eficiencia del árbol, puesto que muchas veces se estaría recorriendo muchos nodos para determinar una altura que ya se había calculado antes, hicimos unas pruebas luego de implementar esta mejora y se comenzó a notar luego de los 20.000 nodos insertados. Gracias a que trabajamos con github, podemos acceder a las versiones viejas y mostrar la diferencia entre las funciones de calcular la altura antes y despues de implementarla como campo de los nodos.
A continuación se muestra el código antes:

```
int avltree_altura(AVLTree arbol){
  int altura = -1; \\ altura arbol vacio.

  if(arbol != NULL){
    altura = max(avltree_altura(arbol->left), avltree_altura(arbol->right)) + 1;
  }

  return altura;
}

```

Esta funcion lo hacía recursivamente, por lo tanto cuando queríamos calcular la altura de una árbol, debíamos recurrir cada subarbol en su completitud.

Luego de hacer el cambio a la estructura, decidimos implementar 2 funciones:

- itree_altura : que devuelve la altura de un arbol. Uno de los motivos por el cual implementamos esta funcion en vez de simplemente escribir arbol−>altura es por nuestra definicion de árbol vacio. Un árbol vacio para nosotros es NULL por lo tanto no tiene campo altura.

```
int itree_altura(ITree arbol){
  // Se define la altura de un arbol vacio como -1.
  int altura = -1;

  if(!itree_es_vacio(arbol))
    altura = arbol->altura;

  return altura;
}

```

- itree_actualizar_altura : que actualiza el campo altura de un árbol.

```
void itree_actualizar_altura (ITree *arbol){
  // La altura del nodo actual va a ser el maximo entre la altura de los hijos
  // mas 1.
  (*arbol)->altura = max2 (itree_altura((*arbol)->left),
    itree_altura((*arbol)->right)) + 1;
}

```

Al encapsular el compartamiento de actualizar la altura, para completar la implementacion, solamente debemos actualizar la altura de un nodo cuando este se pudo haber modificado.
Estas situaciones son:

- Insertar
- Rotar
- Eliminar

Por lo tanto luego de realizar una de estas acciones se debe actualizar la altura. Esta actualizacion es poco costosa por el hecho de que es solo calcular el máximo entre dos numeros y sumarle 1.

5.2. Balanceo

Las mayores dificultades que encontramos en el trabajo fue para balancear los AVLTree. Primeramente, mientras implementabamos la funcion de insercion itree_insertar , veiamos en la teoría que el tipo de rotación requerido dependia del tipo de insercion siendo:

```
1. El elemento X fue insertado en el subárbol izquierdo del hijo izquierdo de N. CASO I-I
2. El elemento X fue insertado en el subárbol derecho del hijo izquierdo de N.   CASO D-I
3. El elemento X fue insertado en el subárbol izquierdo del hijo derecho de N.   CASO I-D
4. El elemento X fue insertado en el subárbol derecho del hijo derecho de N.     CASO D-D
```

Por lo tanto al implementar la funcion de insertar y balancear lo hicimos de la siguiente manera **(código viejo)**[1]:

```
char avltree_insertar (AVLTree *arbol, int dato){
    char mov = ' '; // si fue a la izquierda: i. derecha: d

    if (*arbol == NULL){
        // Agregando nuevo dato.
        AVLTree nuevoSubarbol = malloc (sizeof (AVLTNodo));
        nuevoSubarbol->dato = dato;
        nuevoSubarbol->left = avltree_crear ();
        nuevoSubarbol->right = avltree_crear ();
        *arbol = nuevoSubarbol;
    }
    else {
        char sigMov = ' '; // si fue a la izquierda: i. derecha: d.
        char balanceRequerido[3];
        if (dato < (*arbol)->dato){
          mov = 'i';
          sigMov = avltree_insertar ((&(*arbol)->left), dato);
        } else{
          mov = 'd';
          sigMov = avltree_insertar ((&(*arbol)->right), dato);
        }
        balanceRequerido[0] = mov;
        balanceRequerido[1] = sigMov;
        balanceRequerido[2] = '\0';

        if (!avltree_balanceado (*arbol)){
            *arbol = avltree_balancear (*arbol, balanceRequerido);
        }
    }
return mov;
}

```

Es decir, haciamos que la funcion insertar devuelva un caracter según el último árbol al que se ingreso para continuar la busqueda de la posición correcta para insertar el nuevo nodo.
Luego al momento de que se inserta el nodo, si hubiese un desbalance, sabríamos cual es el caso en el que nos encontramos, y la función de balancear tomaría el string balanceRequerido podria ser: ii , dd, id, di y realizaría las rotaciones necesarias.
Función de balancear **vieja**:

```
AVLTree avltree_balancear (AVLTree arbol, char *caso){
    AVLTree nuevoArbol = NULL;
    // Desbalance hacia afuera, rotacion simple.
    if (!strcmp (caso, "ii"))
        nuevoArbol = avltree_rotacion_der(arbol);
    else if (!strcmp (caso, "dd"))
        nuevoArbol = avltree_rotacion_izq(arbol);
    else if (!strcmp (caso, "di")){
        arbol->right = avltree_rotacion_der(arbol->right);
        nuevoArbol = avltree_rotacion_izq(arbol);
    } else{
        arbol->left = avltree_rotacion_izq(arbol->left);
        nuevoArbol = avltree_rotacion_der(arbol);
    }
    return nuevoArbol;
}

```

[1]Este código es de la etapa en la que estabamos implementando las funciones en AVLTree de enteros

Luego cuando implementamos la función eliminar nos encontramos con muchos problemas tratando de adaptar la funcion de balancear para que funcione, puesto que los casos son un tanto diferentes. Por lo tanto decidimos cambiar totalmenta la forma en la que estabamos balanceando, y por lo tanto, insertando.

Para poder utilizar una misma funcion balancear para ambos casos, es decir para inserción y eliminación, decidimos obtener los casos analizando directamente los factores de balance.
Si un árbol está desbalanceado implica que su *factor de balance es menor a 1 ó mayor a 1*, en el primer caso esta información nos esta diciendo que el hijo izquierdo tiene más altura, luego solo hace falta mirar el *factor de balance* del hijo más alto para determinar que tipo de rotación es necesaria. Es decir, que nodo/s se deben bajar y que cuales se deben subir. Resultando nuestras fuenciones de insertar y balancear de la siguiente forma:

```
ITree itree_balancear (ITree arbol){
    // Si se le pasa un arbol vacio, itree_factor_balance devolveria 0 y no
    // entraria a ninguna otra funcion.
    int factorBalance = itree_factor_balance (arbol);

    // Si la altura del subarbol izquierdo es mayor...
    if (factorBalance > 1){
        // En el caso de balance tras insercion, nunca va a ser igual a 0, pues justamente
        // por algo esta desbalanceado.
        // Ahora bien, cuando estamos balanceando tras eliminacion, puede ser que
        // sea 0, y en ese caso daria lo mismo, por lo que optamos por hacer
        // solo una rotacion simple.
        if (itree_factor_balance (arbol->left) >= 0)
          // Se tiene un caso de izquierda-izquierda, que requiere una rotacion der.
          arbol = itree_rotacion_der (arbol);
        // Si la altura del subarbol derecho del subarbol izquierdo es mayor.
        else {
          // se tiene un caso de izquierda-derecha, que requiere una rotacion doble.
          arbol->left = itree_rotacion_izq (arbol->left);
          arbol = itree_rotacion_der (arbol);
    }
    }
    else if (factorBalance < -1){
        // En el caso de balance tras insercion, nunca va a ser igual a 0, pues justamente
        // por algo esta desbalanceado.
        // Ahora bien, cuando estamos balanceando tras eliminacion, puede ser que
        // sea 0, y en ese caso daria lo mismo, por lo que optamos por hacer
        // solo una rotacion simple.
        if (itree_factor_balance (arbol->right) <= 0){
          // Se tiene un caso de derecha-derecha, que requiere una rotacion izq.
          arbol = itree_rotacion_izq (arbol);
        }
        else {
          // se tiene un caso de derecha-izquierda, que requiere una rotacion doble.
          arbol->right = itree_rotacion_der (arbol->right);
          arbol = itree_rotacion_izq (arbol);
        }
    }

    return arbol;
}

```

Ahora las funciones de eliminar e insertar son void y dentro, se llama a la función de balancear en cada recursión.

```
void itree_insertar (ITree *arbol, Intervalo nIntervalo);
void itree_eliminar_dato (ITree *arbol, Intervalo datoQueEliminar);

```

5.3. Insertar y Eliminar

Las funciones de insertar y eliminar son en su interior funciones de árboles de búsqueda binaria solamente que al final de cada recursión se actualizan los valores que anñadimos para mejora la eficiencia de este tipo de árboles en particular (altura y máxmimo extremo derecho) y se balancean los mismos.

Inserción Es como una inserción estandar de árbol de búsqueda binaria. Esto es, una función recursiva que recorre el árbol valiéndose de una función auxiliar de comparación que servirá como brújula para lograrlo. Hasta que se le pase un árbol vacío, o se tope con el dato que se está queriendo insertar. En el primer escenario, se crea un nuevo nodo para el árbol conteniendo el dato y se inserta. En el segundo, no se hace nada, cerrando el recorrido.
Solo que ademas al final de cada recursion, se balancea el árbol, logrando que sea AVLTree. Además se actualizan los valores de altura y máximo extremo derecho en cada nodo en los que pudo haber cambiado, lo cual es específico de la implementación de árboles de intervalos.

Eliminación También es como una eliminacíon estandar de árboles de búsqueda binaria. Esto sería, una funcion recursiva que recorre el árbol valiéndose de una función auxiliar de comparacion que sirve como brújula para logralo. Hasta que se encuentre el dato a eliminar ó se concluya que no está (esto es si se llega a un árbol vacio). En el primer escenario, hay 3 posibles casos:

- El nodo es una hoja. En este caso se cambia la referencia actual a ese nodo a un árbol vacio'y se libera el espacio.
- El nodo tiene un solo hijo. En este caso, se cambia la referencia al nodo por el hijo y se libera el espacio.
- El nodo tiene ambos hijos. En este caso, decidimos buscar el **mínimo nodo del hijo derecho** puesto que este sería *mayor* a todos los del hijo izquierdo y a su vez, *menor* a todos los del hijo derecho, por lo que pdría bien ocupar la posicion del nodo a eliminar sin implicar cambios. Hubiese sido análogo buscar el máximo en el hijo izquierdo.
  Una vez encontrado, asignamos el valor del dato del minimo al nodo que se deseaba eliminar y luego liberamos el primero.

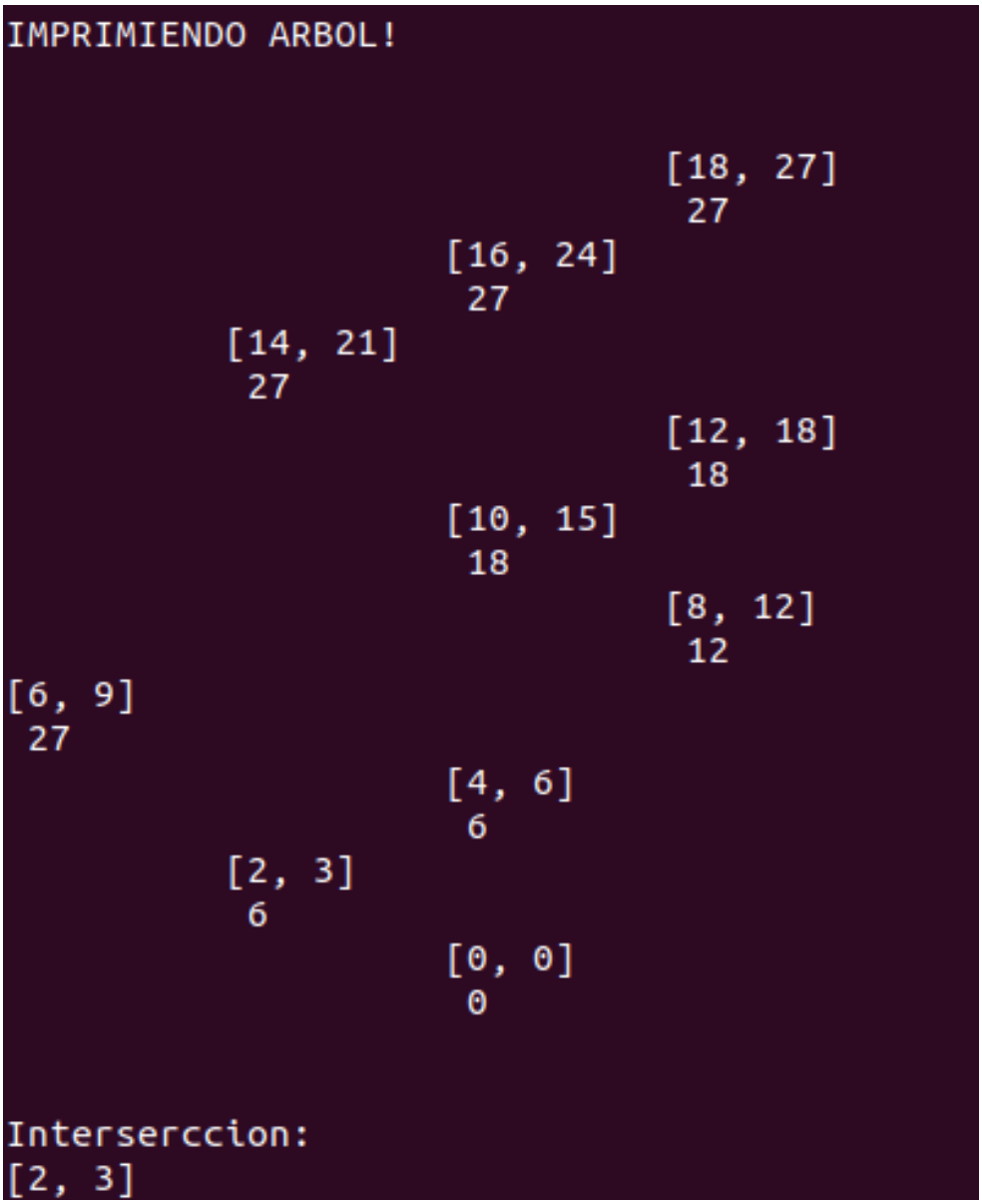
Luego, al igual que al insertar, al final de cada recursión se realizan las mismas acciones de balance y acutalización de datos con los mismos objetivos.

5.4. Instersecar

La función de intersecar, dado un intervalo, recorre un árbol no vacío recursivamente hasta encontrar un nodo que contiene un intervalo que interseca con el parámetro o llegar a un árbol vacío. Recorre con el árbol con el siguiente criterio: Si el hijo izquierdo no es vacío y el extremo izquierdo del intervalo parámetro es menor o igual al máximo extremo derecho del hijo izquierdo, entonces se realiza la recursión sobre el hijo izquierdo. Esto es porque si se cumple la condición, si hay interseccion solo podra haberla en el hijo izquierdo.

```
1 else if(!itree_es_vacio(arbol->left) && intervalo.extIzq <= arbol->left->maxExtDer)
2     resultado = itree_intersecar (arbol->left, intervalo);
3
```

En el siguiente ejemplo, se trata de buscar la interseccion del arbol con el intervalo [3,3]. Notese que el árbol está impreso de costado, debajo de cada intervalo esta el máximo extremo derecho de cada nodo. Esto lo logramos modificando un poco una implementación que encontramos en internet[2].



Por ser un **árbol AVL**, el extremo izquierdo del intervalo del hijo izquierdo es menor o igual al extremo izquierdo del nodo, en el ejemplo: 2 menor que 6.
Y como el extremo izquierdo del intervalo parametro (3), es menor o igual al máximo extremo derecho del hijo izquierdo (6), entonces *sabemos con seguridad de que si existe un intervalo que interseque, está en en ese subárbol.*

[2]Link en la bibliografia

6. Intérprete

6.1. Lectura de la entrada

En cuanto a la lectura de la entrada, probamos varias maneras que fueron generando distintos problemas antes de llegar a la versión final.

El primer intento funcional era con este código:

```
char buffer[80], barra[1];
scanf ("%[^\n]%c", buffer, barra);

```

Esta manera funcionaba siempre y cuando no se ingresara un salto de linea sin haber ingresado ningun otro caracter anteriormente. Para ese caso en particular el programa terminaba ingresando en un bucle en el que seguía tomando una entrada nula infinitamente y no se permitía ingresar más comandos.

La primera solución que encontramos fue con la función fgets:

```
fgets (buffer, sizeof(buffer), stdin);
buffer[strlen(buffer)-1] = '\0';

```

De esta manera solucionamos el problema anterior pero con la desventaja de tener que siempre cortar el último caracter del buffer (ya que fgets toma toda la linea incluido el salto de linea).

Por último llegamos a nuestro código actual:

```
banderaScan = scanf ("%[^\n]", buffer);
getc (stdin);
if (banderaScan == 0)
    printf ("Entrada invalida, ingrese algo.\n");
else{
    ...
}

```

En esta versión el scanf() lee toda la entrada hasta el salto de linea sin incluirlo. Luego getc() lee este salto de linea. Luego, utilizamos el retorno del scanf para aseguranos de que haya algo escrito en la entrada, imprimiendo un mensaje de error en caso contrario. De esta forma solucionamos los problemas de las 2 versiones anteriores.

6.2. Validación de entrada

Luego de que se ingresa el comando, llamamos a una función de validación que nos va a devolver la acción a realizar (y en caso de ser necesario, va a modificar el intervalo que se le pase como parámetro) si el comando es válido.

```
char validar_entrada (char*, Intervalo* );

```

Si el comando no necesita un intervalo podemos hacer una comparación directa para saber si es válido o no (dfs, bfs y salir). Pero para el resto de los comandos es necesario asegurarse de que el intervalo se haya ingresado correctamente.

Para esto hacemos uso de otra función validar_argumento a la que se le pasa la parte de la entrada que contiene el intervalo y se asegura de que este escrito correctamente.

```
int validar_argumento (char *entrada);

```

Por dentro, esta función tiene varias banderas, sirven para asegurarnos de que haya la cantidad correcta de puntos, comas, signos menos, y símbolos één la entrada.
También nos aseguramos de que cada caracter este ubicado correctamente haciendo comparaciones con los caracteres que lo rodean. Por ejemplo, un punto solo puede tener números alrededor suyo, pero a una coma le puede seguir un signo menos o un espacio.

Todas estas características están muy claras en el comentado del código:

```
int coma = 0;      // bandera que indica si se coloco o no una coma
int punto = 0;     // bandera que indica si se coloco o no un punto
int basura = 0;    // caracteres no desesados dentro del intervalo
int espacioM = 0; // espacio mal colocado.
int menos = 0;     // bandera que indica si se coloco o no un signo menos
int exp = 0;       // bandera que indica si hay una e mal colocada
int i = 3;         // contador luego del '[' en la entrada


for (; entrada[i] != ']' && entrada[i] != '\0' && !basura && !espacioM; ++i){
  // Si se ingresa una coma, y esta es valida. Se resetean los contadores de
  // punto y signo menos y se modifica la bandera de coma a 1, lo que
  // implicara que si se ingresa otra coma esta sera invalida.
  if (entrada[i] == ',' && validar_coma (entrada, coma, i)) {
    coma = 1;
    punto = 0;
    menos = 0;
    exp = 0;
  }

  // Si se ingreso un signo menos y es valido. Se modifica la bandera de signo
  // menos a 1, lo que implicara que si se ingresa otro signo menos este sera
  // tomado invalido.
  else if (entrada[i] == '-' && validar_sign_menos (entrada, menos, i))
    menos = 1;

  // Si se ingreso un punto y es valido. Se modifica la bandera de punto a 1,
  // lo que implicara que si se ingresa otro punto este sera tomado invalido.
  else if (entrada[i] == '.' && validar_punto (entrada, punto, i))
    punto = 1;

  // Si se ingreso un espacio en blanco y es valido. Se modifica la bandera de
  // espacio en blanco a 1, lo que implicara que si se ingresa otro espacio en
  // blanco este sera tomado invalido.
  else if (entrada[i] == ' ' && !validar_espacio (entrada, i))
    espacioM = 1;

  else if (entrada[i] == 'e' && validar_exp (entrada, exp, i)){
    exp = 1;
    menos = 0;
    punto = 1;
  }
  // Si se ingreso algo diferente a un espacio o numero (distinguimos el caso
  // de espacio en blanco para identificarlo como un error particular).
  // Se modifica la bandera de basura indicando que se ingreso basura en el
  // intervalo.
  else if (entrada[i] != ' ' && !validar_numero (entrada[i]))
    basura = 1;
}

```

Las funciones validar_exp, validar_espacio, validar_punto, validar_sign_menos, validar_coma todas tienen en cuenta el contexto del programa (es decir el valor actual su bandera) y la posicion del caracter a validar respecto de la entrada. Cada una está tambien bien explciada en el código.
Luego en el programa se puede reconocer con bastante especificidad cuál fue el error de entrada del usuario y se muestra un mensaje de error adecuado.

Entrada válida Una vez se determine que la entada es válida, es decir que el string *entrada*, puede ser escaneado con sscanf(), se determina si los valores del intervalo son válidos, es decir que el extremo izquierdo sea menor o igual que el derecho en nuestro caso.
En caso de no serlo, se devuelve un mensaje de error que explica esto.

7. Bibliografía

Referencias

[1] **Geeksforgeeks** *Imprimir árbol binario en horizontal*, Consultado en Mayo-Junio 2020
`https://www.geeksforgeeks.org/print-binary-tree-2-dimensions/`

[2] **Geeksforgeeks** *Eliminación en AVLTree*, Consultado en Mayo-Junio 2020
`https://www.geeksforgeeks.org/avl-tree-set-2-deletion/?ref=lbp`