Un árbol de búsqueda binaria es la base para la implementación de dos clases de colecciones de biblioteca, "set" y "map". Los árboles en general son muy útiles en las ciencias computacionales.

Preliminares

Un árbol es una colección de nodos, esta colección puede estar vacía; de lo contrario, consta de un nodo distinguido "r" llamado raíz, y cero o más (sub)árboles no vacíos cuya cada raíz está conectada por un borde dirigido desde "r". La raíz de cada subárbol es hija de "r", y r es el padre de la raíz de cada subárbol.

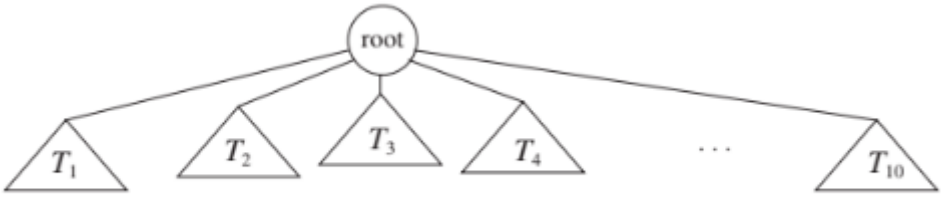


Figure 4.1 Generic tree

Implementación de árboles

Una forma de implementar sería tener en cada nodo un enlace a cada hijo del nodo. Sin embargo, dado que el número de hijos por nodo puede variar tanto que posiblemente exista mucho espacio desperdiciado.

```
1  struct TreeNode
2  {
3      Object    element;
4      TreeNode *firstChild;
5      TreeNode *nextSibling;
6  };
```

Figure 4.3 Node declarations for trees

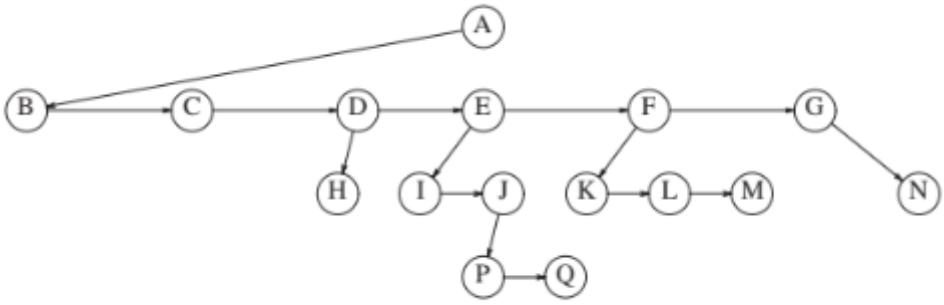


Figure 4.4 First child/next sibling representation of the tree shown in Figure 4.2

Recorrido de árboles con aplicación

Uno de los usos populares es la estructura de directorios en muchos sistemas operativos comunes, como UNIX y DOS.

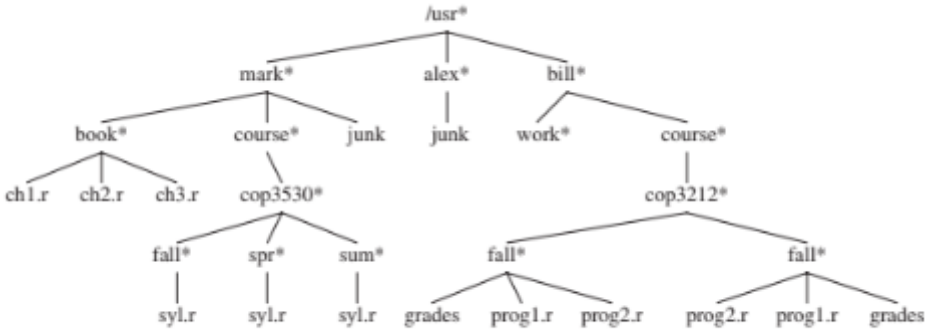


Figure 4.5 UNIX directory

La raíz del directorio de arriba es /usr que contiene tres directorios y ningún archivo normal. El nombre de archivo se obtiene siguiendo tres veces al elemento secundario más a la izquierda. Cada "/" después del primero indica una ventaja; el resultado es la ruta completa.
La función recursiva "listAll" debe iniciarse con una profundidad de 0 para indicar que no hay identificación para la raíz. Esta profundidad es una variable contable interna.
La lógica del algoritmo es que el nombre del objeto de archivo se imprime con el número apropiado de pestañas. Si la entrada es un directorio, procesamos todos los elementos secundarios de forma recursiva, uno por uno.

```
/usr
    mark
        book
            ch1.r
            ch2.r
            ch3.r
        course
            cop3530
                fall
                    syl.r
                spr
                    syl.r
                sum
                    syl.r
        junk
    alex
        junk
    bill
        work
        course
            cop3212
                fall
                    grades
                    prog1.r
                    prog2.r
                fall
                    prog2.r
                    prog1.r
                    grades
```

Figure 4.7 The (preorder) directory listing

Esta estrategia transversal se conoce como recorrido previo al pedido. Donde el trabajo en un nodo se realiza antes de que se procesen (pre) sus hijos. Otro método común para atravesar un árbol es el recorrido posterior al orden. En

un recorrido posterior al orden, el trabajo en un nodo se realiza después de que se evalúen (post) sus hijos.

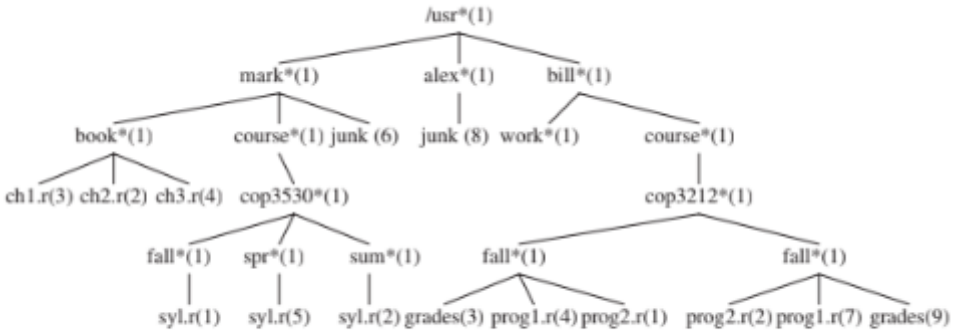


Figure 4.8 UNIX directory with file sizes obtained via postorder traversal

Árbol binario

Es un árbol en el que ningún nodo puede tener más de dos hijos.

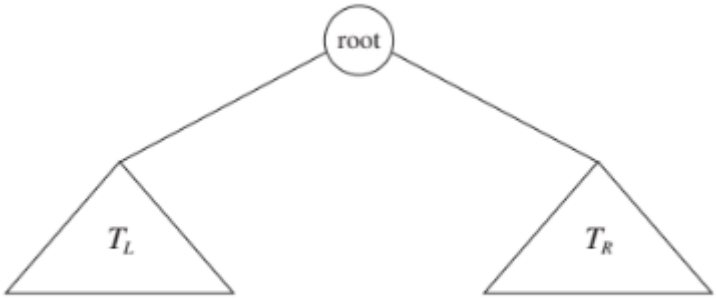


Figure 4.11 Generic binary tree

El árbol binario de arriba consta de una raíz y dos subárboles.

Implementación

Podemos mantener enlaces directos a ellos. La declaración de nodos de árbol es similar en estructura a la de las listas doblemente enlazadas, un nodo es una estructura que consta de la información del elemento más dos punteros (izquierdo y derecho).

```
struct BinaryNode
{
    Object      element;      // The data in the node
    BinaryNode *left;         // Left child
    BinaryNode *right;        // Right child
};
```

Figure 4.13 Binary tree node class (pseudocode)

El árbol de búsqueda ADT: Árboles de búsqueda binarios

Asumiendo que cada nodo del árbol guarda un elemento y que todos son distintivos.
La propiedad que convierte un árbol binario en un árbol de búsqueda binaria es que para cada nodo "x" en el árbol, los valores de todos los elementos en su subárbol izquierdo son menores que el elemento en "x", y los valores de todos los elementos en su subárbol derecho es más grande que el elemento en "x".

Contenedores

Esta operación devuelve "true" si hay un nodo en el árbol "T" que tiene el elemento "X", o "false" si no existe dicho nodo. Si "T" está vacío se regresa falso, si el elemento guardado en "T" es "X" se regresa verdadero y si no es ninguna se hace una llamada recursiva a un subárbol de "T", ya sea hacia la izquierda o hacia la derecha, dependiendo de la relación de "X" con el elemento almacenado en "T".

```
1  /**
2   * Internal method to test if an item is in a subtree.
3   * x is item to search for.
4   * t is the node that roots the subtree.
5   */
6  bool contains( const Comparable & x, BinaryNode *t ) const
7  {
8      if( t == nullptr )
9          return false;
10     else if( x < t->element )
11         return contains( x, t->left );
12     else if( t->element < x )
13         return contains( x, t->right );
14     else
15         return true;    // Match
16 }
```

Figure 4.18 `contains` operation for binary search trees

Hay que notar que es necesario que primero se realice la prueba de un árbol vacío si no se puede generar un error de tiempo de ejecución al intentar acceder a un miembro de datos a través de un puntero nullptr. También hay que notar que as llamadas recursivas son en realidad recursiones de cola y se pueden eliminar fácilmente con un bucle "while".

findMin y findMax

Las rutinas privadas devuelven un puntero al nodo que contiene los elementos más pequeños y más grandes del árbol. Para un "findMin", se comienza en la raíz y se va a la izquierda siempre que quede un hijo, el punto de parada es el elemento más pequeño. Para la rutina "findMax" es la mismo, excepto que la bifurcación es hacia el hijo correcto.

```
/**
 * Internal method to find the smallest item in a subtree t.
 * Return node containing the smallest item.
 */
BinaryNode * findMin( BinaryNode *t ) const
{
    if( t == nullptr )
        return nullptr;
    if( t->left == nullptr )
        return t;
    return findMin( t->left );
}
```

Figure 4.20 Recursive implementation of `findMin` for binary search trees

```
/**
 * Internal method to find the largest item in a subtree t.
 * Return node containing the largest item.
 */
BinaryNode * findMax( BinaryNode *t ) const
{
    if( t != nullptr )
        while( t->right != nullptr )
            t = t->right;
    return t;
}
```

Figure 4.21 Nonrecursive implementation of `findMax` for binary search trees

Insertar

Para insertar "X" en un árbol "T", se procede a bajar por el árbol como con un "contenedor". Si "X" es encontrado no se hace nada, si no se inserta "X" en el último punto del camino recorrido. Para insertar 5, se atraviesa el árbol como si estuviera ocurriendo con un "contain". En el nodo con el elemento 4, se deben ir a la derecha, pero no hay ningún subárbol, por lo que 5 no está en el árbol y este es el lugar correcto para colocar 5.
Los duplicados se pueden manejar manteniendo un campo adicional en el registro del nodo que indique la frecuencia de aparición.

Figure 4.22 Binary search trees before and after inserting 5

Remove

Si el nodo es una hoja, se puede eliminar inmediatamente; si tiene un hijo, el nodo se puede eliminar después de que su padre ajuste un enlace para omitir el nodo. Pero se complica cuando tiene dos hijos la estrategia suele ser reemplazar los datos de este nodo con los datos más pequeños del subárbol derecho y elimine recursivamente ese

nodo, aunque si se espera que el número de eliminaciones sea pequeño se suele usar "lazy deletion" que consiste que cuando se va a eliminar un elemento, se deja en el árbol y simplemente se marca como eliminado.

Destructor y constructor por copia

El destructor llama a "makeEmpty" (que llama una versión recursiva privada), después de procesar los hijos de "T" la llamada para eliminar está hecha para "T". De este modo todos los nodos son reclamados de forma recursiva .
El constructor por copia sigue procedencia usual primero inicializando "root" a "nullptr" y después haciendo una copia de "rhs".

```
1  /**
2   * Destructor for the tree
3   */
4  ~BinarySearchTree( )
5  {
6      makeEmpty( );
7  }
8  /**
9   * Internal method to make subtree empty.
10  */
11 void makeEmpty( BinaryNode * & t )
12 {
13     if( t != nullptr )
14     {
15         makeEmpty( t->left );
16         makeEmpty( t->right );
17         delete t;
18     }
19     t = nullptr;
20 }
```

Figure 4.27 Destructor and recursive `makeEmpty` member function


```
1  /**
2   * Copy constructor
3   */
4  BinarySearchTree( const BinarySearchTree & rhs ) : root{ nullptr }
5  {
6      root = clone( rhs.root );
7  }
8
9  /**
10  * Internal method to clone subtree.
11  */
12 BinaryNode * clone( BinaryNode *t ) const
13 {
14     if( t == nullptr )
15         return nullptr;
16     else
17         return new BinaryNode{ t->element, clone( t->left ), clone( t->right ) };
18 }
```

Figure 4.28 Copy constructor and recursive `clone` member function

Árboles AVL

Un árbol AVL (Adelson-Velskii y Landis) es un árbol de búsqueda binario con una condición de balance, que es fácil de mantener y garantiza que la profundidad del árbol es O (logN). La idea consiste en que los subárboles izquierdo y

derecho tengan la misma altura.

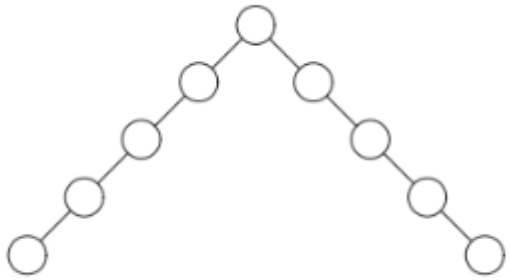


Figure 4.31 A bad binary tree. Requiring balance at the root is not enough.

El árbol AVL es idéntico a un árbol de búsqueda binario, pero para cada nodo del árbol, la altura de los subárboles izquierdo y derecho puede diferir como máximo en 1.

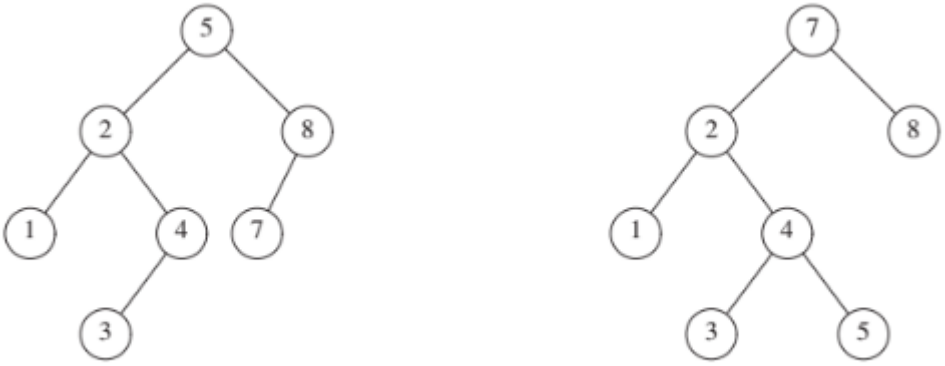


Figure 4.32 Two binary search trees. Only the left tree is AVL.

El árbol de la izquierda es un árbol AVL y el de la derecha no, la información de altura se mantiene para cada nodo y se puede demostrar que la altura de un árbol AVL es como máximo aproximadamente 1,44 log(N + 2) − 1,328, pero en la práctica es sólo un poco más que logN. Para actualizar la información de equilibrio para los nodos en la ruta de regreso a la raíz se realiza las rotación. Luego de una inserción los nodos que se encuentran en la ruta desde el punto de inserción hasta la raíz pueden tener alterado su equilibrio, ahí algún nodo puede violar la condición de los árboles AVL y se puede identificar pues cualquier nodo tiene como máximo dos hijos, y un desequilibrio de altura requiere que las alturas de los dos subárboles de que quieran ser re balanceados difieran en dos. Esto puede ocurrir en cuatro casos diferentes:

1. Una inserción en el subárbol izquierdo del hijo izquierdo del nodo que necesita ser re balanceado.
2. Una inserción en el subárbol derecho del hijo izquierdo del nodo que necesita ser re balanceado.
3. Una inserción en el subárbol izquierdo del hijo derecho del nodo que necesita ser re balanceado.
4. Una inserción en el subárbol derecho del hijo derecho del nodo que necesita ser re balanceado.
   En el primer caso la inserción se ocurre en el “exterior” y se arregla mediante una sola rotación del árbol. En el segundo caso la inserción se produce en el “interior” y se fija mediante una doble rotación.

Rotación simple

La rotación simple que arregla el caso 1 se muestra en la siguiente imagen.

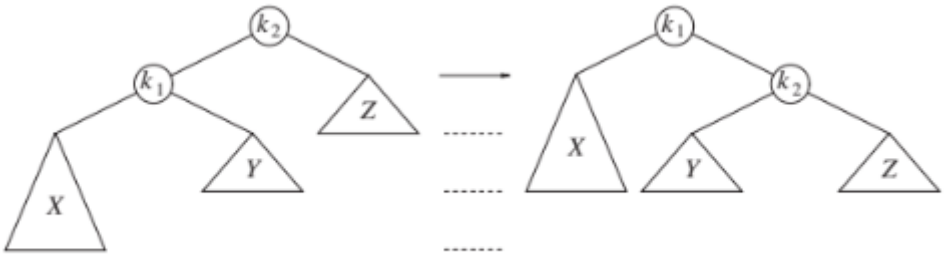


Figure 4.34 Single rotation to fix case 1

El esquema de la izquierda es antes de la rotación y el de la derecha después. El nodo "k2" viola la propiedad de balance AVL porque su subárbol izquierdo es dos niveles más profundo que su subárbol derecho.
Para balancear el árbol se tendría que mover X hacia arriba un nivel y Z hacia abajo. Para hacer esto, reorganizamos los nodos en un árbol equivalente. "X" sube un nivel, "Y" permanece en el mismo nivel y "Z" baja un nivel. "k2" y "k1" no solo cumplen los requisitos de AVL, sino que también tienen subárboles que tienen exactamente la misma altura.

Rotación doble

La rotación doble sirve para resolver problemas como el siguiente:

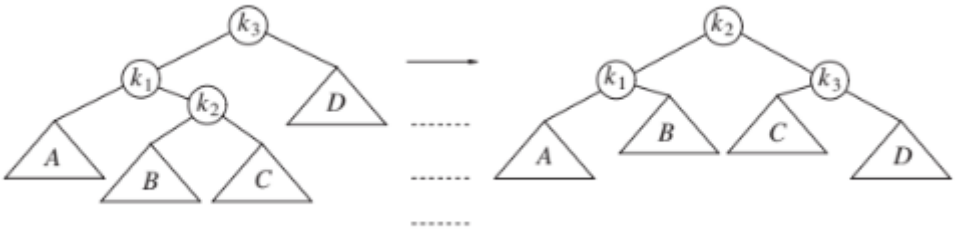


Figure 4.38 Left–right double rotation to fix case 2

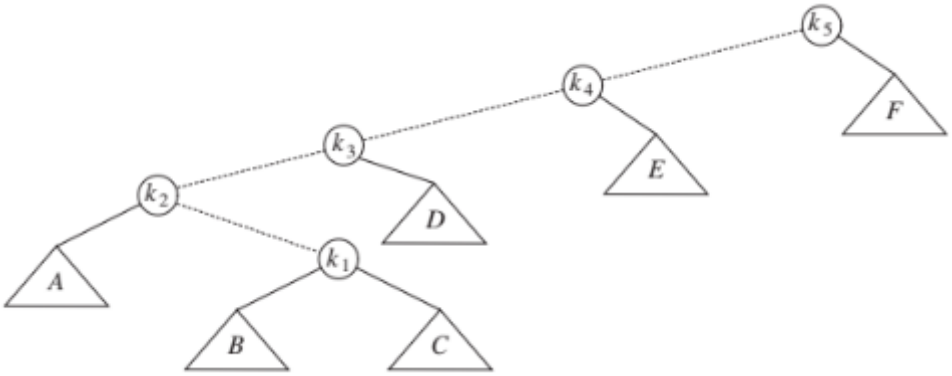
Para re balancear no podemos dejar "k3" como la raíz, así que la alternativa es acomodar "k2" como la nueva raíz. Esto obliga "k1" a ser el hijo izquierdo de "k2" y a "k3" a ser su hijo derecho, y también determina completamente las ubicaciones resultantes de los cuatro subárboles.

Árboles extendidos

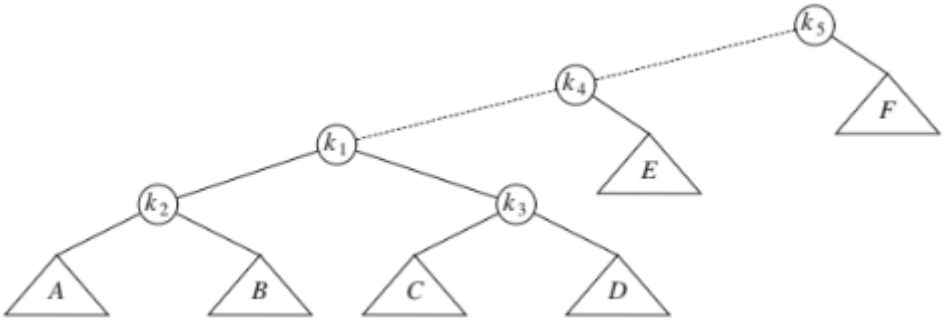
Un árbol extendido es una estructura de datos que garantiza que cualquier operación "M" de árbol consecutiva a partir de un árbol vacío tome como máximo O (M logN) tiempo. Aún existe la posibilidad de que cualquier operación pueda tomar (N) tiempo y, por lo tanto, el límite no sea tan fuerte como un límite O(logN), pero cabe recalcar que no hay malas consecuencias de entrada.
Los árboles extendidos están basados en que el peor de los casos de tiempo O(N) por operaciones no sea tan malo. Pero el problema es que es posible en los árboles de búsqueda binaria que se produzca una secuencia de malos accesos que hacen que se note el tiempo acumulado.
La idea de un árbol extendido es que después de ingresar a un nodo este es empujado a la raíz tras una serie de rotaciones en el árbol AVL.

Expandiendo

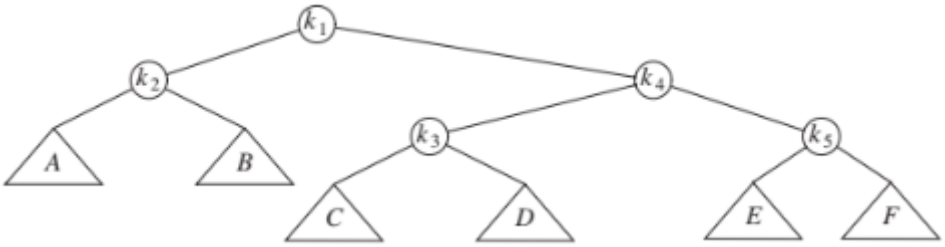
La idea es hacer múltiples rotaciones simples pero siendo selectivos con las rotaciones que se van a hacer. Como ejemplo se tiene el siguiente árbol:



1. El primer caso de separación está en "k1" en el que se realiza una doble rotación AVL (pues es un zigzag) usando "k1", "k2" y "k3".



2. Luego en "k1" se hace una rotación zigzag con "k1", "k4" y "k5". Obteniendo como final:



   Cabe aclarar que el "splaying" también tiene el efecto de reducir aproximadamente a la mitad la profundidad de la mayoría de los nodos en la ruta de acceso.

Recorrido del árbol

La estrategia del recorrido es procesar primero el subárbol izquierdo, luego realizar el procesamiento en el nodo actual y finalmente procesar el subárbol derecho. Yque se realiza un trabajo constante en cada nodo del árbol. Cada nodo se visita una vez y el trabajo realizado en cada nodo consiste en realizar pruebas con nullptr, configurar dos llamadas a funciones y realizar una declaración de salida. Dado que hay trabajo constante por nodo y N nodos, el tiempo de ejecución es O (N). A veces se requiere procesar ambos subárboles primero antes de poder procesar un nodo.

Investigación

Definiciones

- Raíz: Nodo del que derivan los demás nodos.
- Nodo: Elementos de un árbol.
- Nodo terminal: Nodo que no contiene ningún sub árbol.

- Profundidad: Número máximo de nodos en una rama.
- Árbol binario: Conjunto finito de cero cumpliendo las condiciones de que existe un nodo denominado raíz del árbol y que cada nodo puede tener 0, 1 ó 2 subárboles, conocidos como subárbol izquierdo y subárbol derecho.
- Recorrido: Proceso que permite acceder de una sola vez a cada uno de los nodos del árbol.

Opinión

El capítulo empezó siendo sencillo pero en el momento de ver las diferentes formas de rotar y el cómo se hacían las rotaciones empezó a ser un poco más complejo, de igual forma en el libro hay cosas que no quedaban del todo claras así que en algunas cosas Juan y Dani nos hacían el favor de explicar cosas en el Discord. El tema general fue interesante sobre todo porque creo que los árboles en los videojuegos tienen ciertas aplicaciones como por ejemplo los juegos estilo "Until Dawn" o "Detroit: Become Human" en el que según las elecciones cambia la historia a mi particularmente me gustan ese tipo de juegos.

Bibliografía

-“Arboles en C++ – Programación”. Programación. Accedido el 9 de noviembre de 2023. [En línea]. Disponible: https://www.programacion.com.py/escritorio/c/arboles-en-c