Universidad Tecnológica de Pereira

CAPÍTULO: ÁRBOLES BINARIOS DE BÚSQUEDA (BINARY SEARCH TREE)
CURSO DE ESTRUCTURA DE DATOS - Código IS304

Programa de Ingeniería de Sistemas y Computación
Profesor Hugo Humberto Morales Peña
Semestre Agosto/Diciembre de 2021

Árbol binario de búsqueda

Un árbol binario de búsqueda está organizado, como su nombre lo indica, en un árbol binario, como se muestra en la figura 1. Éste árbol puede ser representado por una estructura de datos enlazada, donde cada nodo es un objeto. Además de un campo llave (*key*) y de otros posibles campos de información, cada nodo contiene campos $left$, $right$ y p, que apuntan a los nodos correspondientes a su hijo izquierdo, derecho y su padre, respectivamente. Si uno de los hijos o el padre, no existen, el campo respectivo contiene el valor `NIL`. El nodo raíz es el único nodo en el árbol cuyo apuntador al padre es `NIL`.

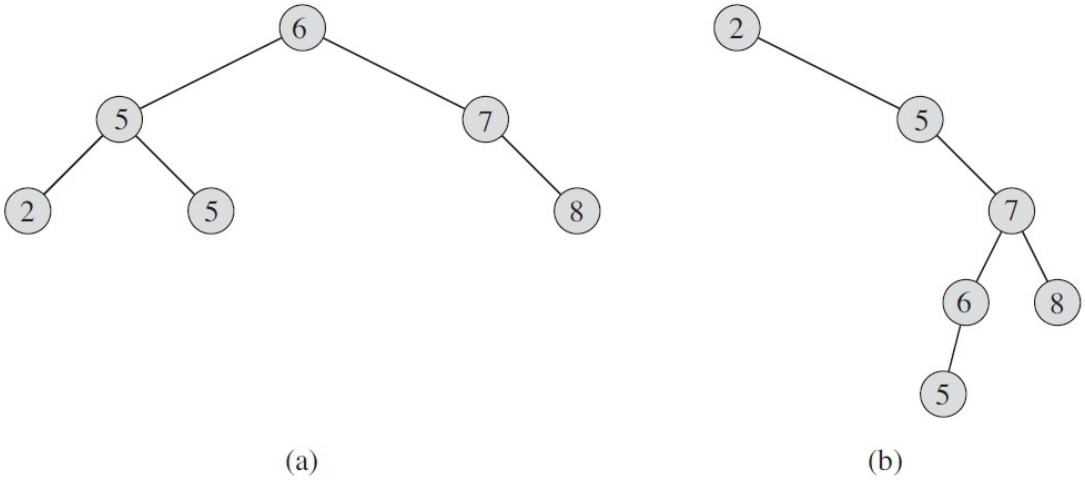

Figura 1: Árbol binario de búsqueda. Para cualquier nodo x, las llaves en el subárbol izquierdo de x son estrictamente menores a la llave de x, y las llaves en el subárbol derecho de x son mayores o iguales a la llave de x. Árboles binarios de búsqueda diferentes pueden representar el mismo conjunto de valores. El peor caso en tiempo de ejecución para las operaciones sobre el árbol binario de búsqueda es proporcional a la altura del árbol. (a) Un árbol binario de búsqueda sobre 6 nodos con altura 2. (b) Un árbol binario de búsqueda menos eficiente con altura 4 que contiene las mismas llaves.

Las llaves en un árbol binario de búsqueda siempre son guardadas en una forma tal que satisfaga **la propiedad de un árbol binario de búsqueda**:

Sea x un nodo en el árbol binario de búsqueda. Si y es un nodo en el subárbol a la izquierda de x, entonces $key[y] < key[x]$. Si y es un nodo en el subárbol a la derecha de x, entonces $key[x] \leq key[y]$.

La propiedad del árbol binario de búsqueda, permite imprimir en orden todas las llaves de un árbol binario de búsqueda, mediante un algoritmo recursivo llamado **recorrido del árbol en-orden**. Este algoritmo es llamado así debido a que la llave de la raíz de un subárbol es impresa entre los valores del subárbol a su izquierda y los valores del subárbol a su derecha. (De manera similar, un **recorrido del árbol en pre-orden** imprime la raíz antes de los valores de sus subárboles, y un **recorrido del árbol en post-orden** imprime la raíz después de los valores en sus subárboles.) Para usar el siguiente procedimiento para imprimir todos los elementos de un árbol binario de búsqueda T, se llama a `INORDER-TREE-WALK(`$root[T]$`)`

function `INORDER-TREE-WALK(` x `)`

```
1   if x ≠ NIL then
2       INORDER-TREE-WALK(left[x])
3       print key[x]
4       INORDER-TREE-WALK(right[x])
```

Ejercicio:

Hacer el paso a paso del algoritmo `INORDER-TREE-WALK` sobre el árbol binario de búsqueda (a) de la Figura 1. El paso a paso del recorrido en orden se ilustra en las Figuras 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19 y 20:

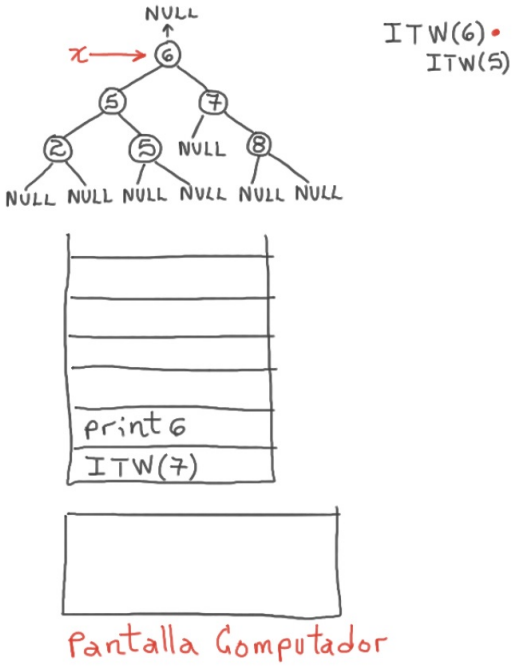

Figura 2: Recorrido en orden - llamado recursivo con el nodo raíz (nodo de información 6).

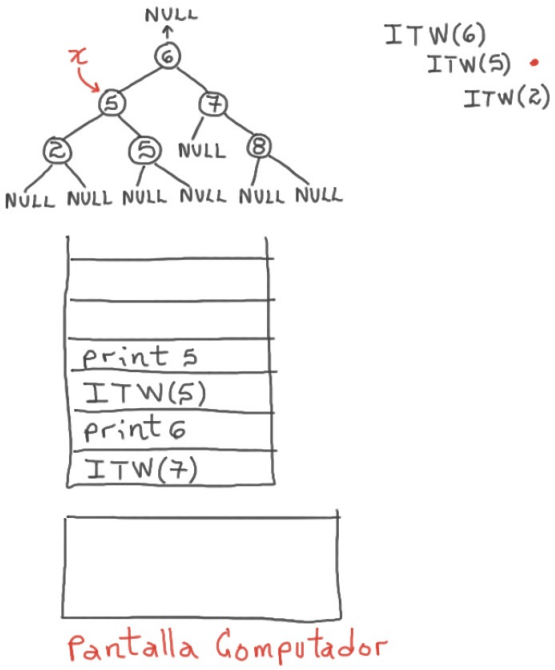

Figura 3: Recorrido en orden - llamado recursivo con el primer nodo de información 5.

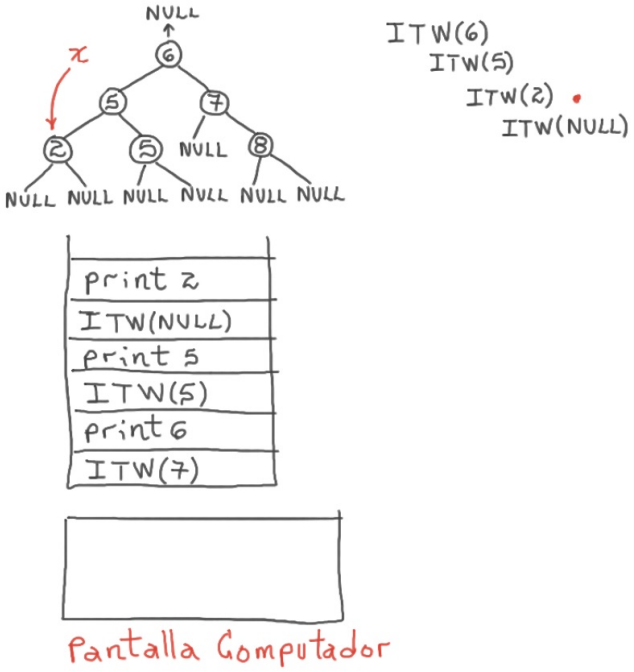

Figura 4: Recorrido en orden - llamado recursivo con el nodo de información 2.

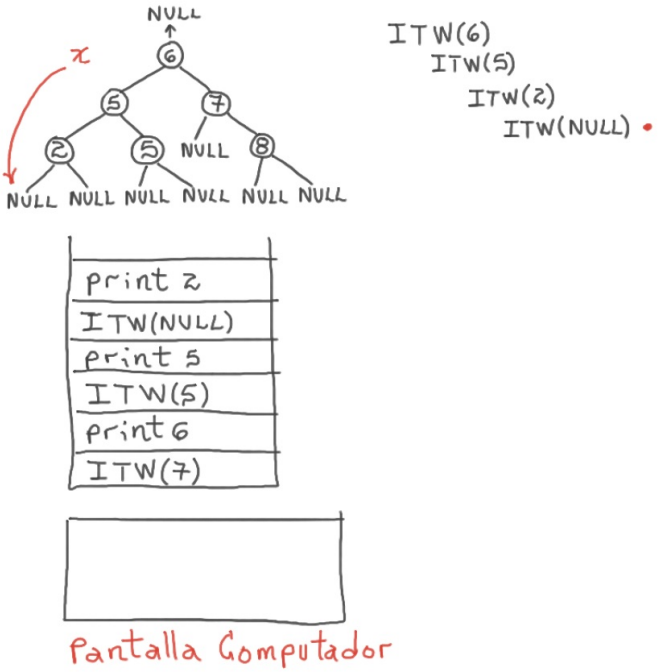

Figura 5: Recorrido en orden - llamado recursivo con el hijo por la izquierda del nodo de información 2.

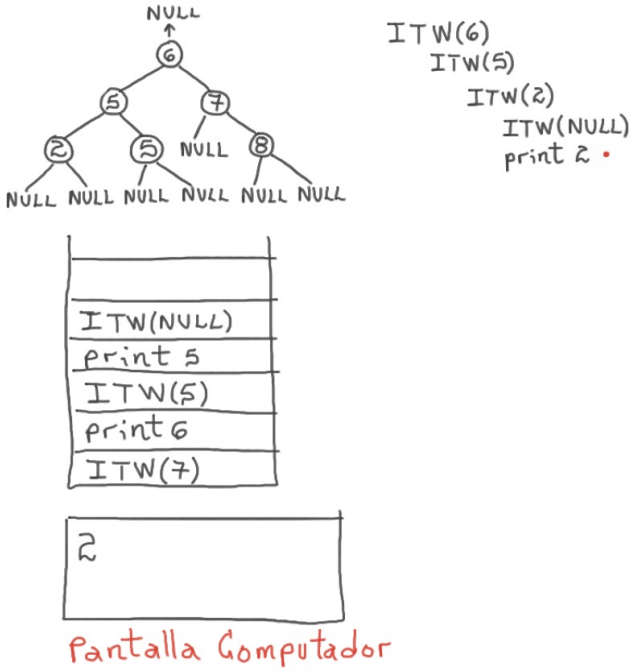

Figura 6: Recorrido en orden - impresión del 2 por pantalla.

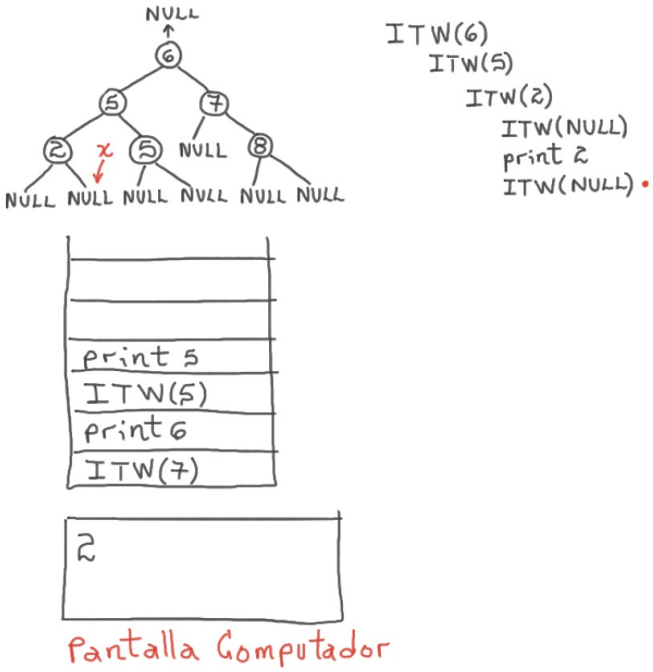

Figura 7: Recorrido en orden - llamado recursivo con el hijo por la derecha del nodo de información 2.

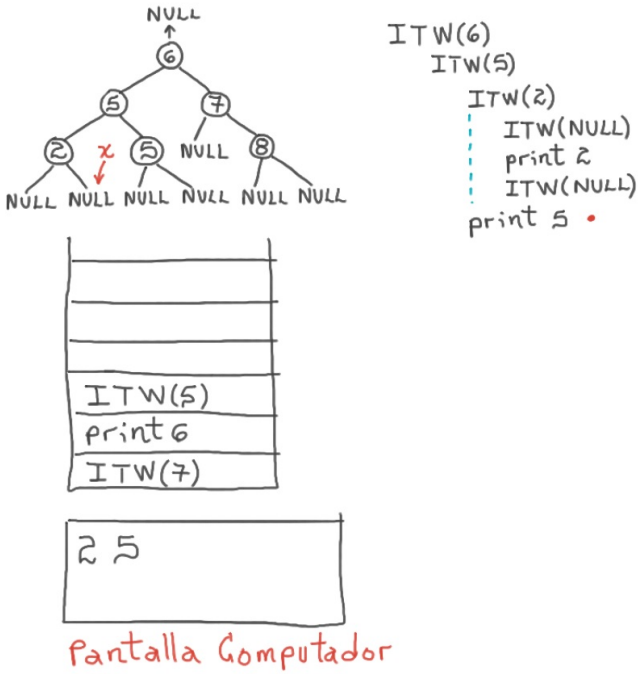

Figura 8: Recorrido en orden - impresión del 5 por pantalla.

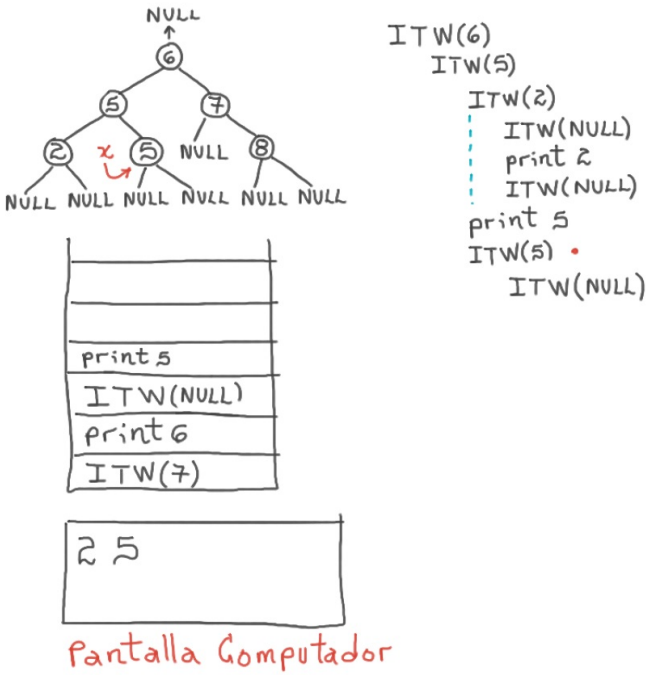

Figura 9: Recorrido en orden - llamado recursivo con el hijo por derecha del primer nodo de información 5.

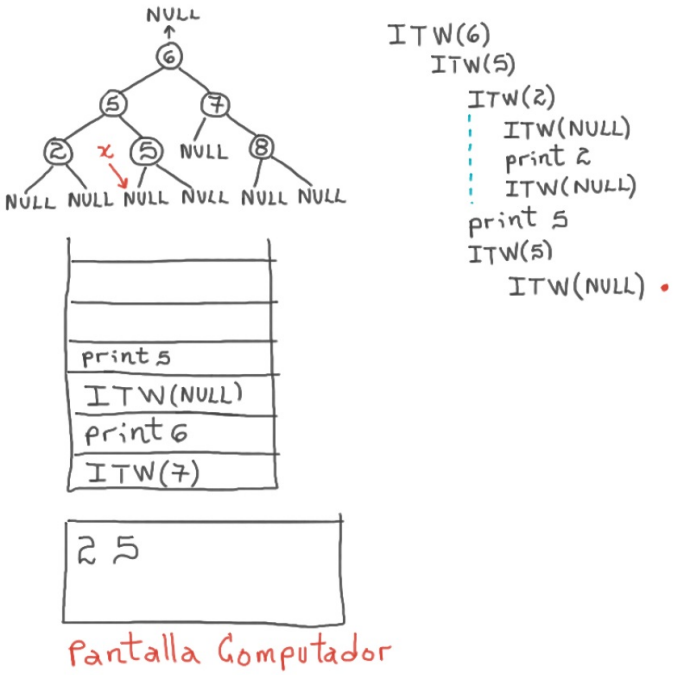

Figura 10: Recorrido en orden - llamado recursivo con el hijo por izquierda del segundo nodo de información 5.

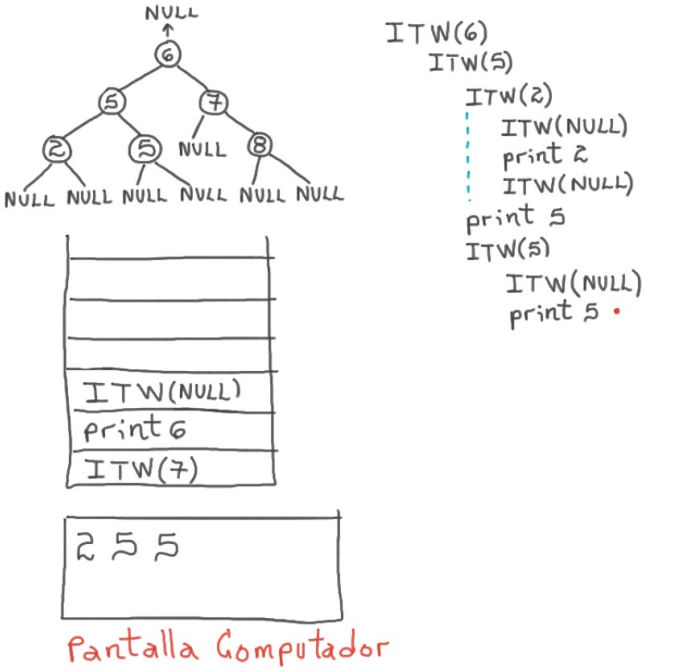

Figura 11: Recorrido en orden - impresión del segundo 5 por pantalla.

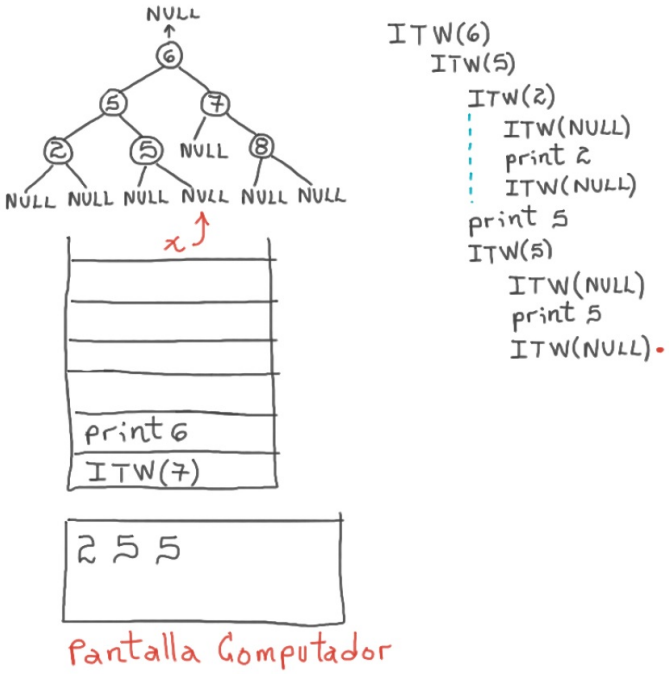

Figura 12: Recorrido en orden - llamado recursivo con el hijo por derecha del segundo nodo de información 5.

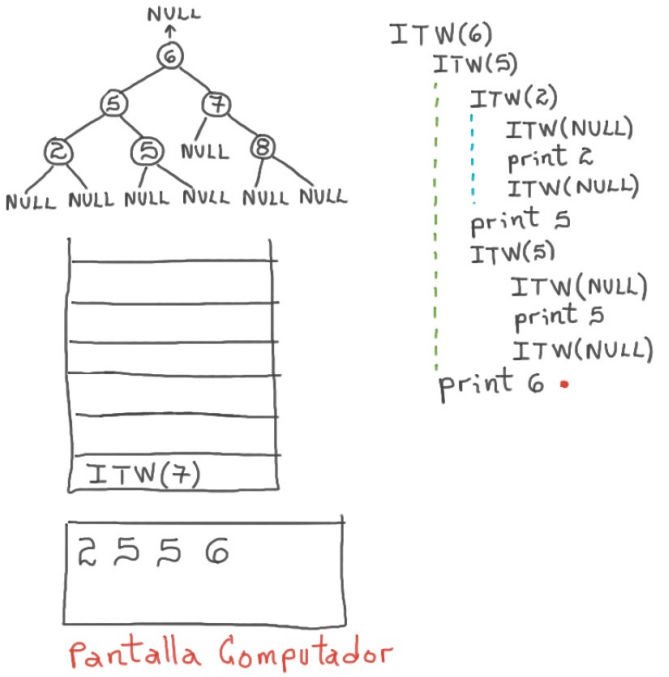

Figura 13: Recorrido en orden - impresión del 6 por pantalla.

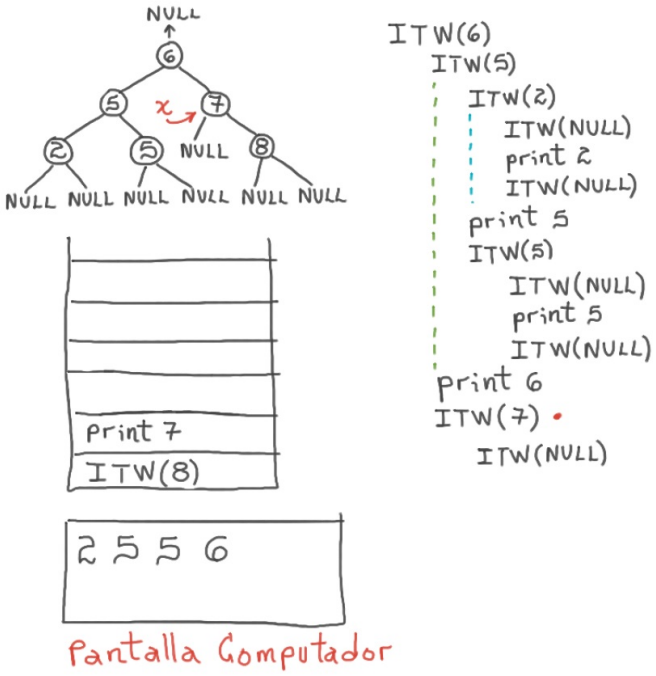

Figura 14: Recorrido en orden - llamado recursivo con el hijo por derecha del nodo de información 6.

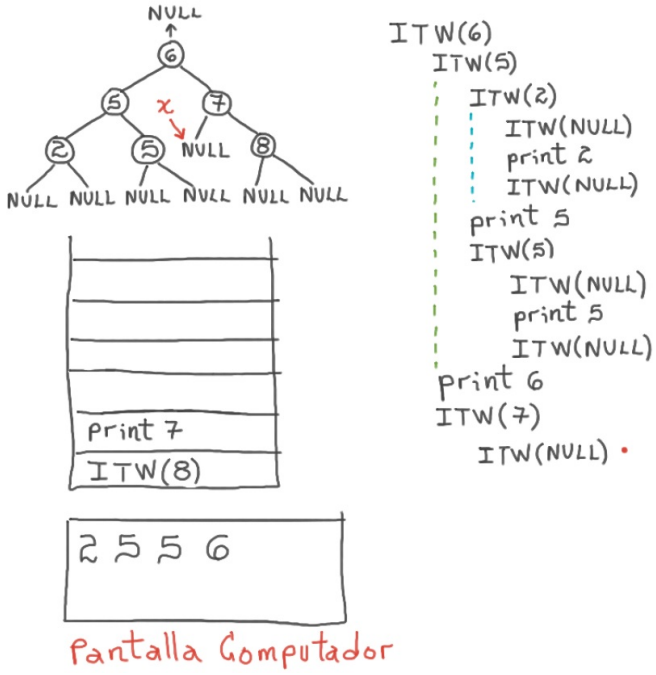

Figura 15: Recorrido en orden - llamado recursivo con el hijo por izquierda del nodo de información 7.

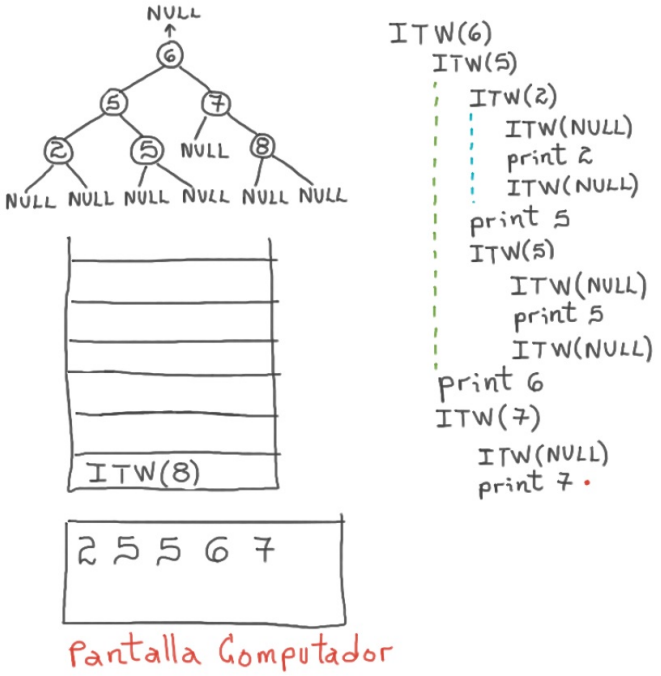

Figura 16: Recorrido en orden - impresión del 7 por pantalla.

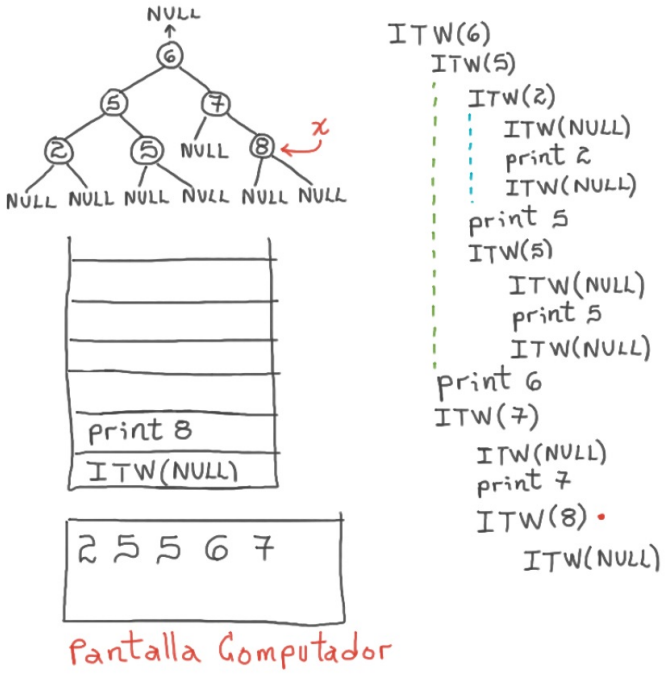

Figura 17: Recorrido en orden - llamado recursivo con el hijo por derecha del nodo de información 7.

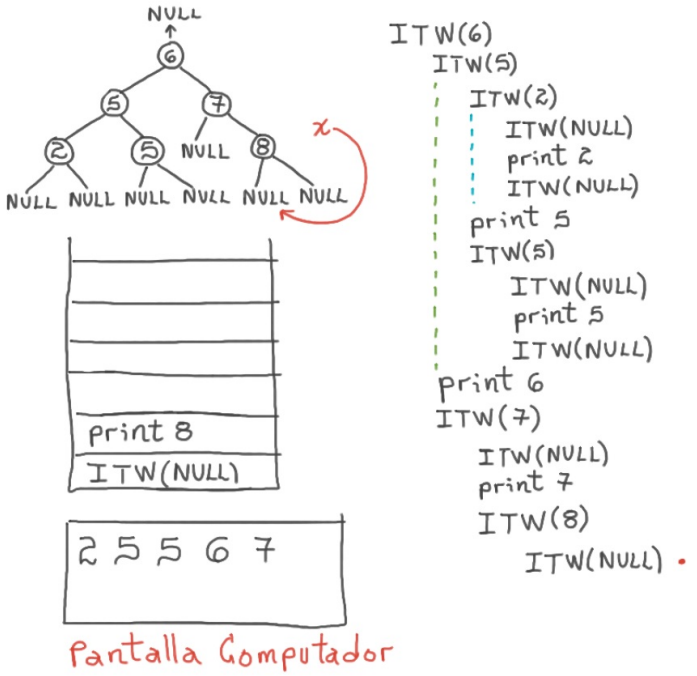

Figura 18: Recorrido en orden - llamado recursivo con el hijo por izquierda del nodo de información 8.

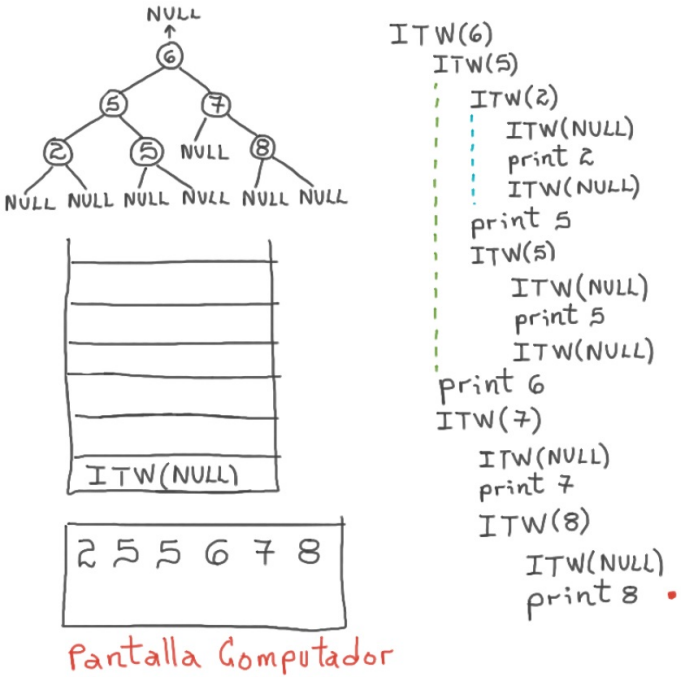

Figura 19: Recorrido en orden - impresión del 8 por pantalla.

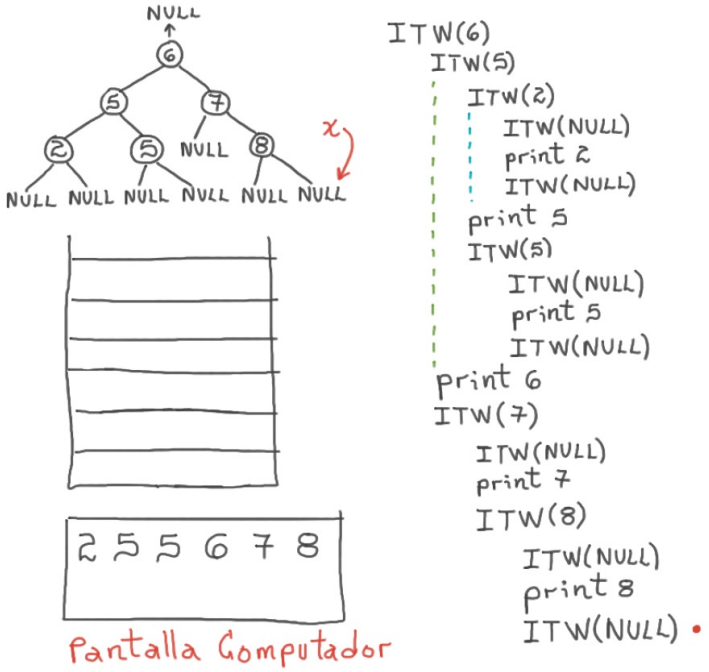

Figura 20: Recorrido en orden - llamado recursivo con el hijo por derecha del nodo de información 8.

Consultar un árbol binario de búsqueda

Una operación común realizada en un árbol binario de búsqueda, es localizar una llave guardada en el árbol. Además de la operación de búsqueda (`SEARCH`), los árboles binarios de búsqueda pueden apoyar consultas tales como el mínimo elemento (`MINIMUM`), el máximo (`MAXIMUM`), el predecesor (`PREDECESSOR`) y el sucesor (`SUCCESSOR`).

Búsqueda

La siguiente función busca un nodo con una llave dada, en el árbol binario de búsqueda. Dados un puntero a la raíz del árbol y una llave k, `TREE-SEARCH` retorna un puntero al nodo con la llave k si existe; de otro modo, retorna `NIL`.

```
function TREE-SEARCH( x, k )
1   if x == NIL  or k == key[x] then
2       return x
3   if k < key[x] then
4       return TREE-SEARCH(left[x], k)
5   else
6       return TREE-SEARCH(right[x], k)
```

La misma función, puede ser escrita iterativamente "simulando" la recursión con un ciclo **while**. En la mayoría de computadores, esta versión es más eficiente.

function ITERATIVE-TREE-SEARCH(x, k)

1 **while** $x \neq$ NIL **and** $k \neq key[x]$ **do**

2 **if** $k < key[x]$ **then**

3 $x \leftarrow left[x]$

4 **else**

5 $x \leftarrow right[x]$

6 **return** x

Mínimo y Máximo

Un elemento en un árbol binario de búsqueda, cuya llave sea la mínima, siempre se puede encontrar siguiendo los punteros izquierdos desde la raíz hasta que se encuentre un NIL. La siguiente función retorna un puntero al elemento más pequeño en el subárbol cuya raíz esta dada por el nodo x.

function TREE-MINIMUM(x)

1 **while** $left[x] \neq$ NIL **do**

2 $x \leftarrow left[x]$

3 **return** x

El pseudocódigo para TREE-MAXIMUM es simétrico.

function TREE-MAXIMUM(x)

1 **while** $right[x] \neq$ NIL **do**

2 $x \leftarrow right[x]$

3 **return** x

Sucesor y predecesor

Dado un nodo en un árbol binario de búsqueda, resulta importante en algunos casos ser capaz de encontrar su sucesor en el orden determinado por el recorrido en-orden del árbol. Si todas las llaves son distintas, el sucesor de un nodo x es el nodo con la menor llave mas grande que $key[x]$. La estructura de un árbol binario de búsqueda permite determinar el sucesor de un nodo sin comparar llaves. La siguiente función retorna el sucesor de un nodo x en un árbol binario de búsqueda, si este existe, y NIL si x tiene llave más grande en el árbol.

function TREE-SUCCESSOR(x)

1 **if** $right[x] \neq$ NIL **then**

2 **return** TREE-MINIMUM($right[x]$)

3 $y \leftarrow p[x]$

4 **while** $y \neq$ NIL **and** $x == right[y]$ **do**

5 $x \leftarrow y$

6 $y \leftarrow p[y]$

7 **return** y

El algoritmo para el predecesor (TREE-PREDECESSOR), es simétrico.

function TREE-PREDECESSOR(x)

1 **if** $left[x] \neq$ NIL **then**

2 **return** TREE-MAXIMUM($left[x]$)

3 $y \leftarrow p[x]$

4 **while** $y \neq$ NIL **and** $x == left[y]$ **do**

5 $x \leftarrow y$

6 $y \leftarrow p[y]$

7 **return** y

Inserción y borrado

Las operaciones de inserción y borrado causan que el conjunto dinámico representado por un árbol binario de búsqueda cambie. La estructura de datos, debe ser modificada para reflejar este cambio, pero de una forma de modo que la propiedad del árbol binario de búsqueda se siga cumpliendo. Modificar el árbol para insertar un elemento nuevo es relativamente sencillo, pero manejar el borrado es un poco más complicado.

Inserción

Se usa el procedimiento TREE-INSERT para insertar un nuevo valor k en el árbol binario de búsqueda T. Se crea un nodo nuevo z, en el cual se almacena el valor k, con este nodo nuevo se modifica el árbol y algunos de los campos de z, de forma que z es insertado en una posición apropiada en el árbol.

function `TREE-INSERT`(T, k)

```
1   Make node z
2   key[z] ← k
3   left[z] ← NIL
4   right[z] ← NIL
5   y ← NIL
6   x ← T
7   while x ≠ NIL do
8       y ← x
9       if key[z] < key[x] then
10          x ← left[x]
11      else
12          x ← right[x]
13  p[z] ← y
14  if y == NIL then
15      T ← z   //El árbol T está vacío
16  else
17      if key[z] < key[y] then
18          left[y] ← z
19      else
20          right[y] ← z
21  return T
```

La Figura 21 muestra como `TREE-INSERT` trabaja. Como los procedimientos `TREE-SEARCH` e `ITERATIVE-TREE-SEARCH`, `TREE-INSERT` empieza en la raíz del árbol y traza un camino hacia abajo. El puntero x traza el camino, y el puntero y es mantenido como el padre de x. Después de la inicialización, el ciclo while en las líneas 7-12 causa que estos dos punteros se muevan abajo en el árbol, moviéndose de izquierda a derecha dependiendo de la comparación de $key[z]$ con $key[x]$, hasta que x apunte a `NIL`. Este `NIL` ocupa la posición donde se quiere ubicar el elemento de entrada z. Las líneas 13-20 cambian los punteros que causan que z sea insertado.

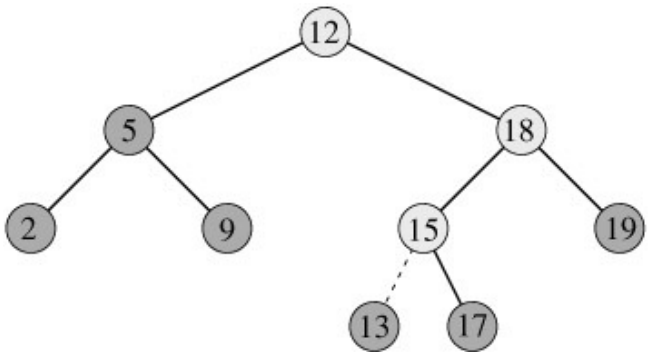

Figura 21: Insertando un elemento con llave 13 en un árbol binario de búsqueda. Los nodos ligeramente sombreados indican el camino desde la raíz hasta la posición donde el elemento es insertado. La línea punteada indica el enlace del árbol que es añadido para insertar el elemento.

Ejemplo 1:

Hacer el paso a paso del algoritmo `TREE-INSERT(T, 19)` sobre el árbol binario de búsqueda de la Figura 22.

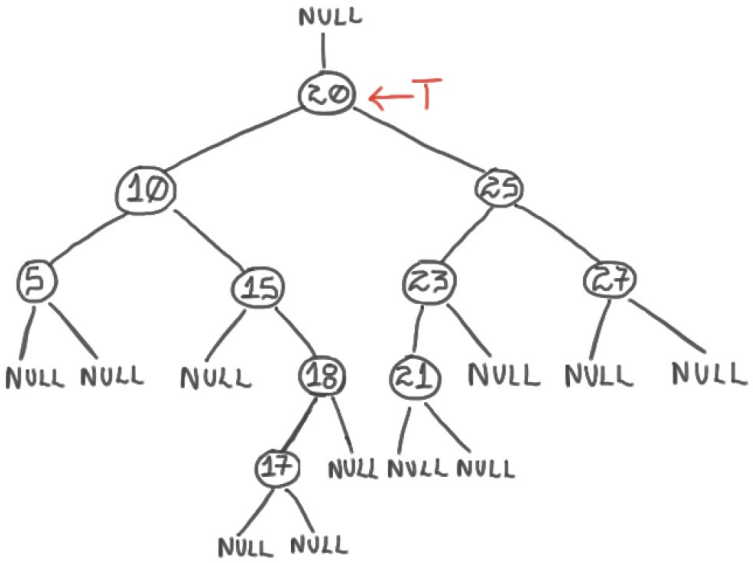

Figura 22: Árbol binario de búsqueda sobre el cual se debe insertar el número 19.

El árbol binario de búsqueda de resultado después de la inserción es el que se encuentra en la Figura 23.

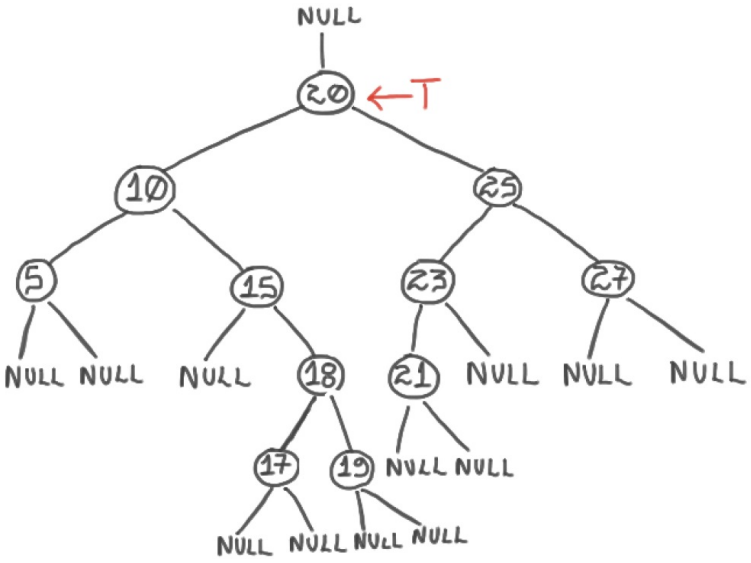

Figura 23: Árbol binario de búsqueda con el número 19 insertado.

Ejercicio 1:

Hacer el paso a paso del algoritmo `TREE-INSERT(T, 22)` sobre el árbol binario de búsqueda de la Figura 24.

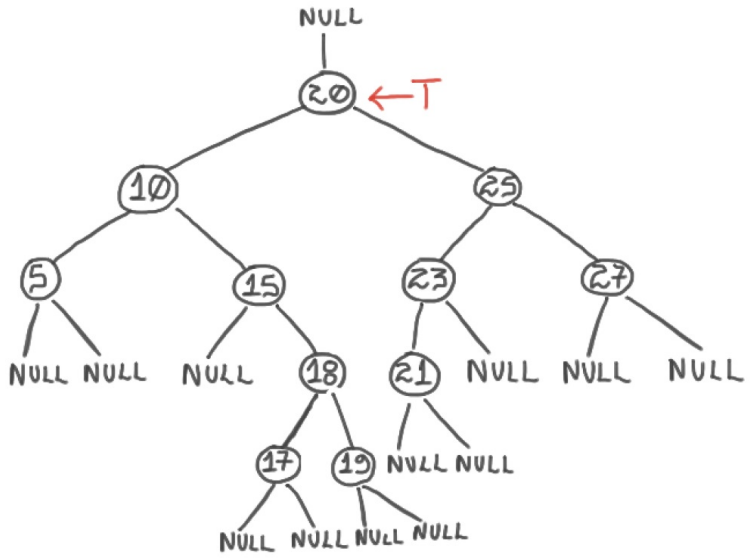

Figura 24: Árbol binario de búsqueda sobre el cual se debe insertar el número 22.

El árbol binario de búsqueda de resultado después de la inserción es el que se encuentra en la Figura 25.

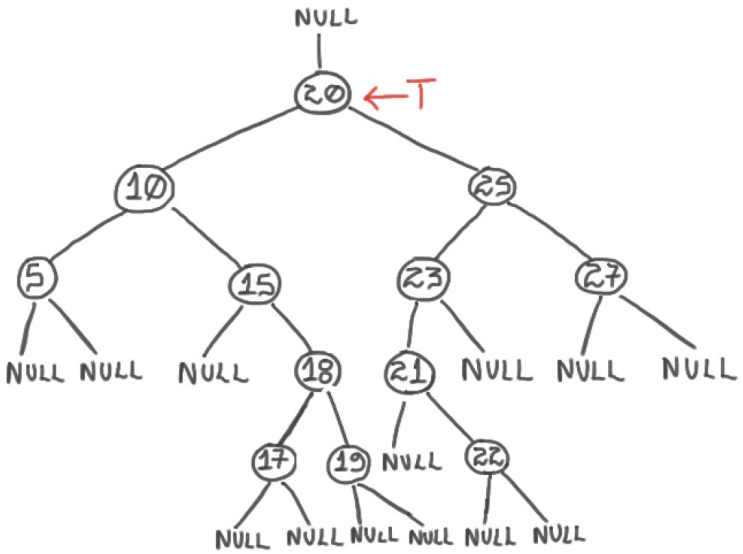

Figura 25: Árbol binario de búsqueda con el número 22 insertado.

Borrado

La función para borrar un nodo z de un árbol binario de búsqueda toma como argumento un puntero a z. La función considera los 3 casos mostrados en la Figura 26. Si z no tiene hijos, se modifica su padre $p[z]$ para reemplazar z con `NIL` como su hijo. Si el nodo tiene solo un hijo, se hace que empalme con z haciendo un nuevo enlace entre su hijo y su padre. Finalmente, si el nodo tiene 2 hijos, se empalma el sucesor y de z, el cual no tiene hijo izquierdo y se reemplaza la llave de z (y todos los posibles campos de información), con los de y.

```
function TREE-DELETE( T, z )
1   if left[z] == NIL or right[z] == NIL then
2       y ← z
3   else
4       y ← TREE-SUCCESSOR( z )
5   if left[y] ≠ NIL then
6       x ← left[y]
7   else
8       x ← right[y]
9   if x ≠ NIL then
10      p[x] ← p[y]
11  if p[y] == NIL then
12      T ← x
13  else
14      if y == left[p[y]] then
15          left[p[y]] ← x
16      else
17          right[p[y]] ← x
18  if y ≠ z then
19      key[z] ← key[y]
20      Copy all information fields from y to z
21  FREE( y )
22  return T
```

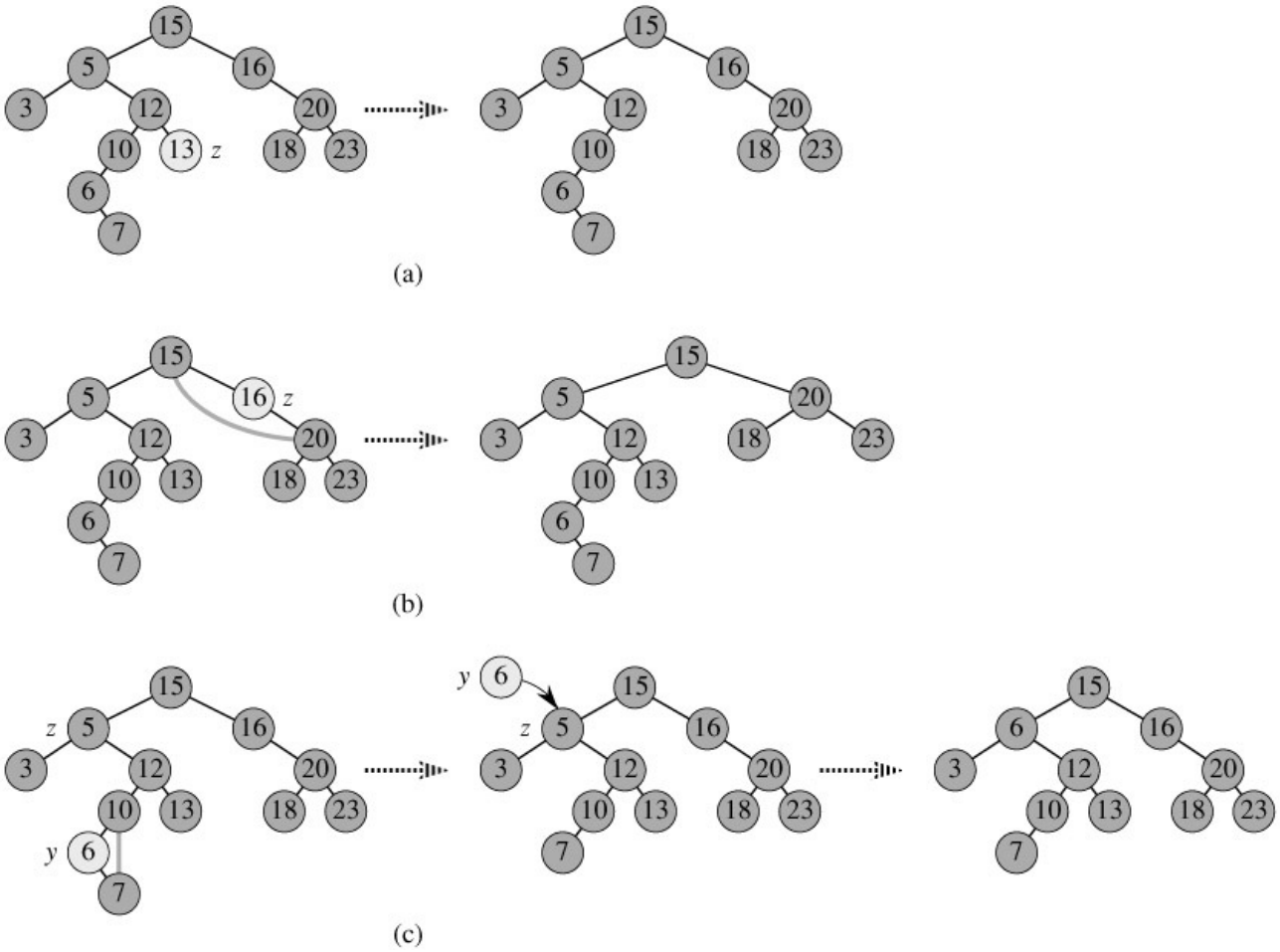

Figura 26: Borrando un nodo z de un árbol binario de búsqueda. Donde el nodo es removido dependiendo de cuantos hijos tenga z; este nodo es mostrado ligeramente sombreado. (a) Si z no tiene hijos, se remueve simplemente. (b) Si z tiene un solo hijo, se empalma con z. (c) Si z tiene 2 hijos, se empalma con su sucesor y, el cual tiene al menos un hijo, y luego se reemplaza la llave de z (y todos los posibles campos de información), con los de y.

Ejemplo 2:

Hacer el paso a paso del algoritmo `TREE-DELETE(T, z)` sobre el árbol binario de búsqueda de la Figura 27.

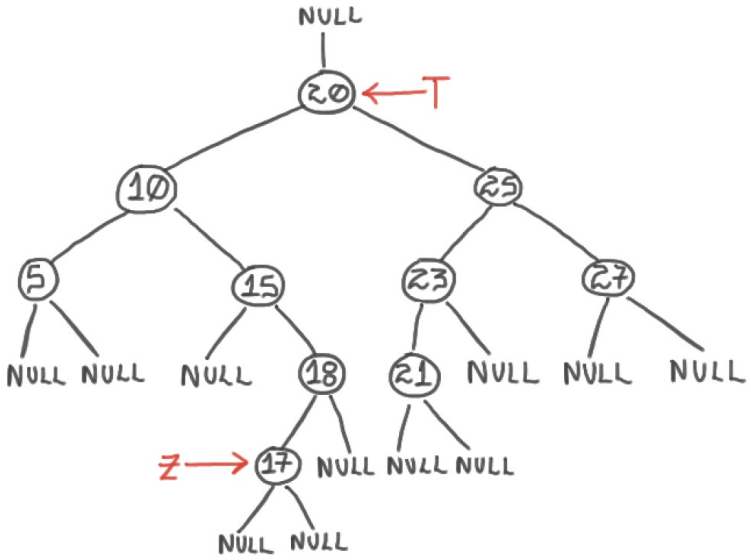

Figura 27: Árbol binario de búsqueda sobre el cual se debe borrar el número 17.

El árbol binario de búsqueda de resultado después del borrado es el que se encuentra en la Figura 28.

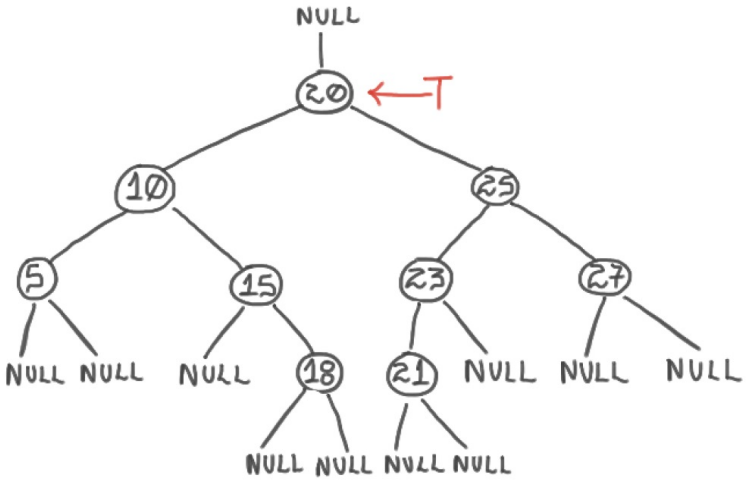

Figura 28: Árbol binario de búsqueda sin el número 17.

Ejemplo 3:

Hacer el paso a paso del algoritmo `TREE-DELETE(T, z)` sobre el árbol binario de búsqueda de la Figura 29.

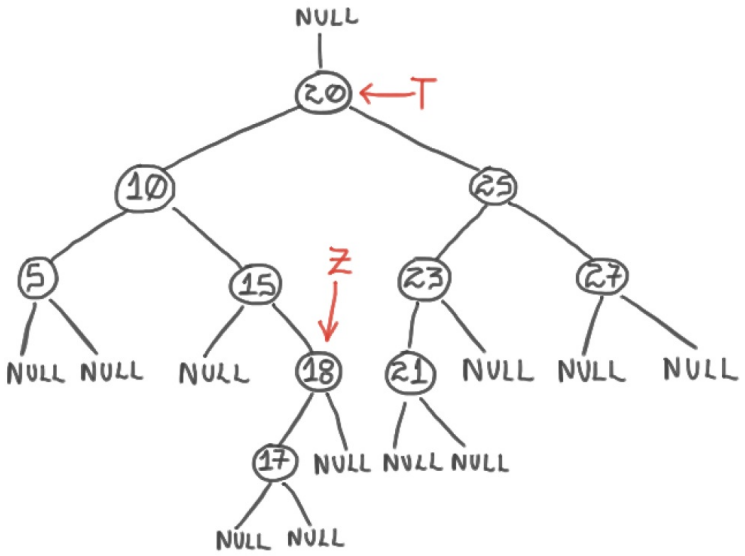

Figura 29: Árbol binario de búsqueda sobre el cual se debe borrar el número 18.

El árbol binario de búsqueda de resultado después del borrado es el que se encuentra en la Figura 30.

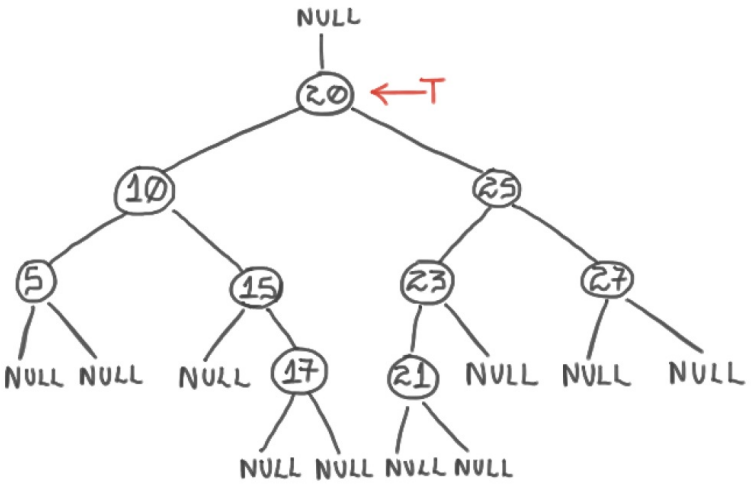

Figura 30: Árbol binario de búsqueda sin el número 18.

Ejemplo 4:

Hacer el paso a paso del algoritmo `TREE-DELETE(T, z)` sobre el árbol binario de búsqueda de la Figura 31.

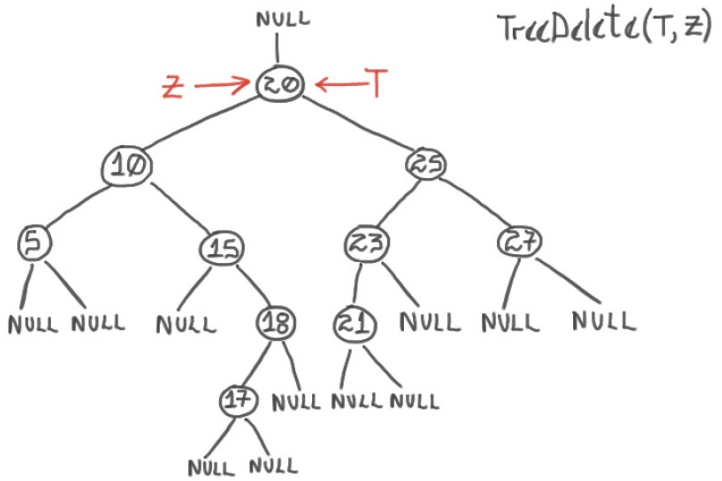

Figura 31: Árbol binario de búsqueda sobre el cual se debe borrar el número 20.

El árbol binario de búsqueda de resultado después del borrado es el que se encuentra en la Figura 32.

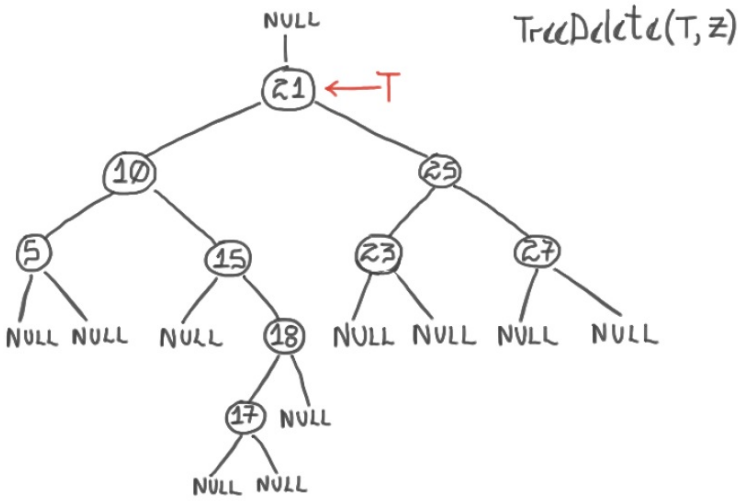

Figura 32: Árbol binario de búsqueda sin el número 20.

Ejercicio 2:

Hacer el paso a paso del algoritmo `TREE-DELETE(T, z)` sobre el árbol binario de búsqueda de la Figura 33.

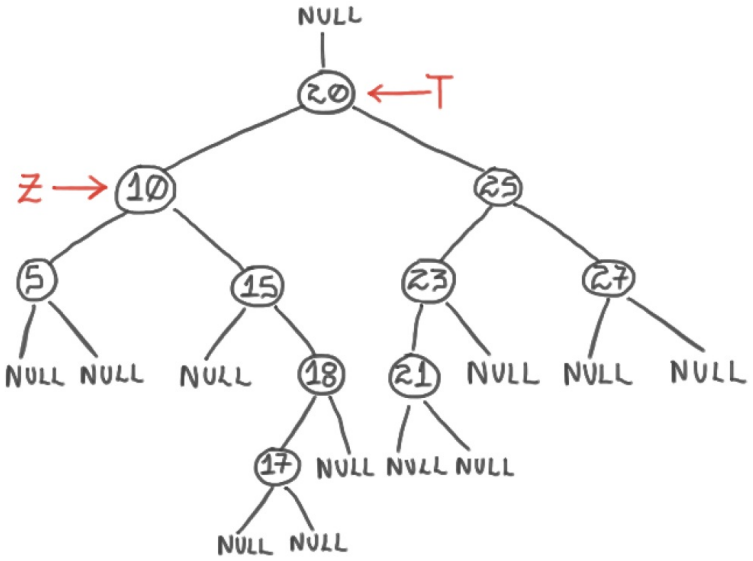

Figura 33: Árbol binario de búsqueda sobre el cual se debe borrar el número 10.

El árbol binario de búsqueda de resultado después del borrado es el que se encuentra en la Figura 34.

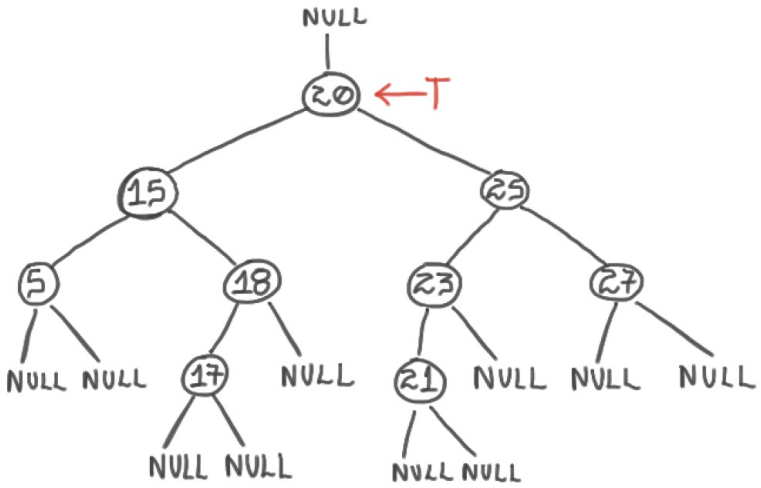

Figura 34: Árbol binario de búsqueda sin el número 10.

Programa Completo para el Manejo de Árboles Binarios de Búsqueda

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

struct nodeTree
```

```
{
    int key;
    struct nodeTree *left;
    struct nodeTree *right;
    struct nodeTree *p;
};

void InorderTreeWalk(struct nodeTree *x)
{
    if(x != NULL)
    {
        InorderTreeWalk(x->left);
        printf("%d ", x->key);
        InorderTreeWalk(x->right);
    }
}

struct nodeTree *TreeSearch(struct nodeTree *x, int k)
{
    while((x != NULL) && (k != x->key))
    {
        if(k < x->key)
            x = x->left;
        else
            x = x->right;
    }

    return x;
}

struct nodeTree *TreeMinimum(struct nodeTree *x)
{
    while(x->left != NULL)
        x = x->left;

    return x;
}

struct nodeTree *TreeMaximum(struct nodeTree *x)
{
    while(x->right != NULL)
        x = x->right;

    return x;
}

struct nodeTree *TreeSuccessor(struct nodeTree *x)
{
    struct nodeTree *y;

    if(x->right != NULL)
        return TreeMinimum(x->right);

    y = x->p;
    while((y != NULL) && (x == y->right))
    {
        x = y;
        y = y->p;
    }

    return y;
}

struct nodeTree *TreePredecessor(struct nodeTree *x)
{
    struct nodeTree *y;

    if(x->left != NULL)
```

```
        return TreeMaximum(x->left);

    y = x->p;
    while((y != NULL) && (x == y->left))
    {
        x = y;
        y = y->p;
    }

    return y;
}

struct nodeTree *TreeInsert(struct nodeTree *T, int k)
{
    struct nodeTree *x, *y, *z;

    z = (struct nodeTree *)malloc(sizeof(struct nodeTree));
    z->key = k;
    z->left = NULL;
    z->right = NULL;
    y = NULL;
    x = T;

    while(x != NULL)
    {
        y = x;
        if(z->key < x->key)
            x = x->left;
        else
            x = x->right;
    }

    z->p = y;
    if(y == NULL)
        T = z; /* Empty tree. */
    else
    {
        if(z->key < y->key)
            y->left = z;
        else
            y->right = z;
    }

    return T;
}

struct nodeTree *TreeDelete(struct nodeTree *T, struct nodeTree *z)
{
    struct nodeTree *x, *y;

    if((z->left == NULL) || (z->right == NULL))
        y = z;
    else
        y = TreeSuccessor(z);

    if(y->left != NULL)
        x = y->left;
    else
        x = y->right;

    if(x != NULL)
        x->p = y->p;

    if(y->p == NULL)
        T = x;
    else
    {
        if(y == y->p->left)
```

```
            y->p->left = x;
        else
            y->p->right = x;
    }

    if(y != z)
    {
        z->key = y->key;
        /* Copy all information fields from y to z. */
    }

    free(y);
    return T;
}

int main()
{
    int operation, element;
    struct nodeTree *T, *z;
    T = NULL;

    while(scanf("%d %d", &operation, &element) != EOF)
    {
        if(operation == 1) /* Insert */
        {
            T = TreeInsert(T, element);
            InorderTreeWalk(T);
            printf("\n");
        }
        else
        {
            if(operation == 2) /* Delete */
            {
                z = TreeSearch(T, element);
                if(z == NULL)
                {
                    printf("The %d is not in the tree\n", element);
                    InorderTreeWalk(T);
                    printf("\n");
                }
                else
                {
                    T = TreeDelete(T, z);
                    InorderTreeWalk(T);
                    printf("\n");

                    if(T != NULL)
                        printf("key[T]: %d\n", T->key);
                }
            }
            else
                printf("Bad use. \n 1 ... Insert \n 2 ... Delete \n");
        }
    }

    return 0;
}
```

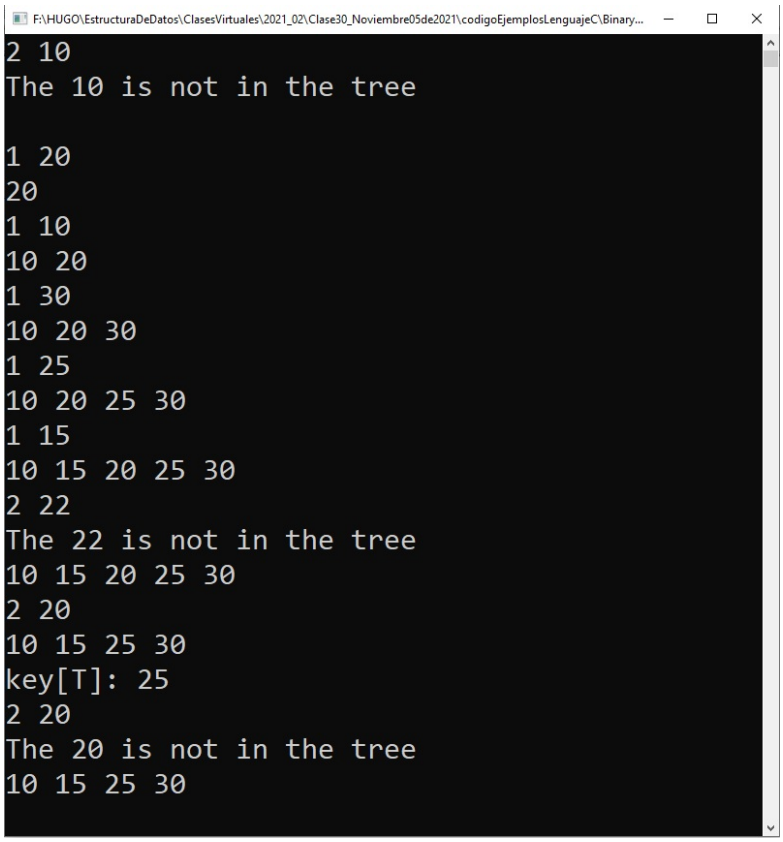

Figura 35: Salida del programa para el mantenimiento de árboles binarios de búsqueda.

Programming Challenge: "Add All - Sumar Todos"[1]

Si!, el nombre del problema refleja el trabajo a realizar; es justamente sumar un conjunto de números. Usted puede sentirse motivado a escribir un programa en su lenguaje de programación favorito para sumar todo el conjunto de números. Tal problema debe ser algo muy sencillo para su nivel de programación. Por este motivo le vamos a poner un poco de sabor y vamos a volver el problema mucho más interesante.

Ahora, la operación de suma tiene un costo, y el costo es el resultado de sumar dos números. Por lo tanto, el costo de sumar 1 y 10 es 11. Por ejemplo, si usted quiere sumar los números 1, 2 y 3, hay tres formas de hacerlo:

- Forma 1:
  $1 + 2 = 3$, costo $= 3$
  $3 + 3 = 6$, costo $= 6$
  Costo total $= 9$
- Forma 2:
  $1 + 3 = 4$, costo $= 4$
  $2 + 4 = 6$, costo $= 6$
  Costo total $= 10$
- Forma 3:
  $2 + 3 = 5$, costo $= 5$
  $1 + 5 = 6$, costo $= 6$
  Costo total $= 11$

[1] https://www.hackerrank.com/contests/utp-open-2018/challenges/add-all-2-1

Yo creo que usted ya ha entendido su misión, sumar un conjunto de números enteros tal que el costo de la operación suma sea mínimo.

Input specification:

La entrada contiene múltiples casos de prueba. Cada caso de prueba debe comenzar con un número entero positivo N ($2 \leq N \leq 5000$), seguido por N números enteros positivos (todos son mayores o iguales a 1 y menores o iguales a 100000). La entrada es finalizada por un caso donde el valor de N es igual a cero. Este caso no debe ser procesado.

Output specification:

Para cada caso de prueba imprimir una sola línea con un número entero positivo que representa el costo total mínimo de la operación suma después de sumar los N números.

Example input:

```
3
1 2 3
4
1 2 3 4
5
5 4 3 2 1
0
```

Example output:

```
9
19
33
```

Solución del reto "Add All - Sumar Todos"

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

struct nodeTree
{
    int key;
    struct nodeTree *left;
    struct nodeTree *right;
    struct nodeTree *p;
};

void InorderTreeWalk(struct nodeTree *x)
{
    if(x != NULL)
    {
        InorderTreeWalk(x->left);
        printf("%d ", x->key);
        InorderTreeWalk(x->right);
    }
}

struct nodeTree *TreeSearch(struct nodeTree *x, int k)
{
    while((x != NULL) && (k != x->key))
    {
        if(k < x->key)
```

```
            x = x->left;
        else
            x = x->right;
    }

    return x;
}

struct nodeTree *TreeMinimum(struct nodeTree *x)
{
    while(x->left != NULL)
        x = x->left;

    return x;
}

struct nodeTree *TreeMaximum(struct nodeTree *x)
{
    while(x->right != NULL)
        x = x->right;

    return x;
}

struct nodeTree *TreeSuccessor(struct nodeTree *x)
{
    struct nodeTree *y;

    if(x->right != NULL)
        return TreeMinimum(x->right);

    y = x->p;
    while((y != NULL) && (x == y->right))
    {
        x = y;
        y = y->p;
    }

    return y;
}

struct nodeTree *TreePredecessor(struct nodeTree *x)
{
    struct nodeTree *y;

    if(x->left != NULL)
        return TreeMaximum(x->left);

    y = x->p;
    while((y != NULL) && (x == y->left))
    {
        x = y;
        y = y->p;
    }

    return y;
}

struct nodeTree *TreeInsert(struct nodeTree *T, int k)
{
    struct nodeTree *x, *y, *z;

    z = (struct nodeTree *)malloc(sizeof(struct nodeTree));
    z->key = k;
    z->left = NULL;
    z->right = NULL;
    y = NULL;
    x = T;
```

```
    while(x != NULL)
    {
        y = x;
        if(z->key < x->key)
            x = x->left;
        else
            x = x->right;
    }

    z->p = y;
    if(y == NULL)
        T = z; /* Empty tree. */
    else
    {
        if(z->key < y->key)
            y->left = z;
        else
            y->right = z;
    }

    return T;
}

struct nodeTree *TreeDelete(struct nodeTree *T, struct nodeTree *z)
{
    struct nodeTree *x, *y;

    if((z->left == NULL) || (z->right == NULL))
        y = z;
    else
        y = TreeSuccessor(z);

    if(y->left != NULL)
        x = y->left;
    else
        x = y->right;

    if(x != NULL)
        x->p = y->p;

    if(y->p == NULL)
        T = x;
    else
    {
        if(y == y->p->left)
            y->p->left = x;
        else
            y->p->right = x;
    }

    if(y != z)
    {
        z->key = y->key;
        /* Copy all information fields from y to z. */
    }

    free(y);
    return T;
}

int main()
{
    int n, element, index;
    long long int result;
    struct nodeTree *T, *z;

    while(scanf("%d", &n) && (n > 0))
```

```
    {
        T = NULL;
        for(index = 1; index <= n; index++)
        {
            scanf("%d", &element);
            T = TreeInsert(T, element);
        }

        result = 0;

        for(index = 1; index < n; index++)
        {
            z = TreeMinimum(T);
            element = z->key;
            T = TreeDelete(T, z);

            z = TreeMinimum(T);
            element += z->key;
            T = TreeDelete(T, z);

            result += element;
            T = TreeInsert(T, element);
        }

        free(T);
        printf("%lld\n", result);
    }

    return 0;
}
```

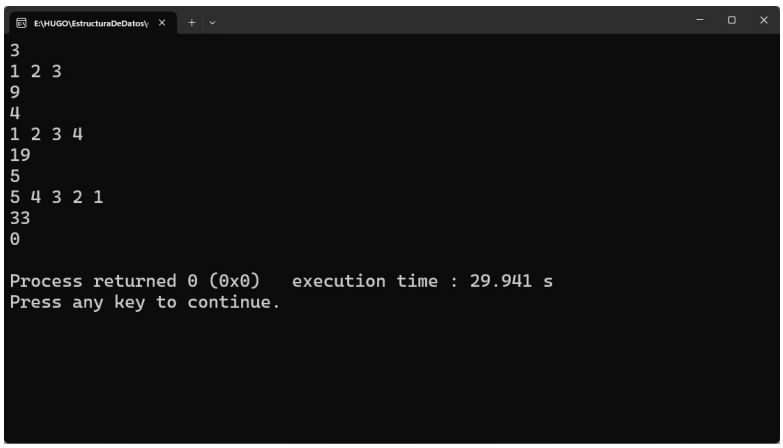

Figura 36: Salida del programa para el reto “Add All”.