EBook Gratis

APRENDIZAJE data-structures

Free unaffiliated eBook created from **Stack Overflow contributors.**

#data-structures

Tabla de contenido

Acerca de ... 1

Capítulo 1: Empezando con estructuras de datos ... 2

Observaciones ... 2

Examples ... 2

Introducción a las estructuras de datos ... 2

Array: una estructura de datos simple ... 2

Capítulo 2: Apilar ... 4

Examples ... 4

Introducción a la pila ... 4

Usando pilas para encontrar palíndromos. ... 5

Implementación de la pila utilizando una matriz y una lista enlazada ... 6

Comprobación de paréntesis equilibrados ... 8

Capítulo 3: Árbol de búsqueda binaria ... 10

Examples ... 10

Creando un nodo en BST ... 10

insertando un nodo en el árbol de búsqueda binario ... 10

Capítulo 4: Árbol de segmentos ... 12

Examples ... 12

Introducción al árbol de segmentos ... 12

Implementación de Segment Tree Using Array ... 15

Realizar una consulta de rango mínimo ... 20

Propagación perezosa ... 23

Capítulo 5: Cola ... 31

Examples ... 31

Introducción a la cola ... 31

Implementación de la cola mediante el uso de matriz. ... 31

Implementación de una cola circular. ... 33

Representación de lista enlazada de la cola ... 34

Capítulo 6: Deque (doble cola de cola) ... 37

Examples ... 37

Inserción y eliminación tanto del frente como del final de la cola....37
Capítulo 7: Estructura de datos de búsqueda de la unión....38
Introducción....38
Examples....38
Teoría....38
Implementacion basica....39
Mejoras: Compresión de ruta....40
Mejoras: Unión por tamaño....41
Mejoras: Unión por rango....41
Mejora final: Unión por rango con almacenamiento fuera de los límites....42
Capítulo 8: Lista enlazada....44
Examples....44
Introducción a las listas enlazadas....44
Lista enlazada individualmente....44
Lista enlazada XOR....45
¿Por qué esto se llama la lista enlazada de memoria eficiente?....45
¿Es esto diferente de una lista doblemente vinculada?....45
¿Como funciona?....46
Código de ejemplo en C....48
Referencias....50
Omitir lista....50
¿Es esto diferente de una lista enlazada individualmente?....50
¿Como funciona?....50
Referencias....51
Lista doblemente vinculada....51
Un ejemplo básico de SinglyLinkedList en Java....52
Capítulo 9: Matrices....56
Observaciones....56
Examples....56
Introducción a las matrices....56
Capítulo 10: Montón binario....57

Introducción 57
Examples 57
Ejemplo 57
Capítulo 11: Recorridos de grafos 60
Introducción 60
Examples 60
Primera búsqueda de profundidad 60
Amplia primera búsqueda 61
Capítulo 12: Trie (árbol de prefijo / árbol de radix) 63
Examples 63
Introducción a Trie 63
Implementación de Trie 67
Creditos 69

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: data-structures

It is an unofficial and free data-structures ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official data-structures.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con estructuras de datos

Observaciones

Esta sección proporciona una descripción general de qué son las estructuras de datos y por qué un desarrollador puede querer usarlas.

También debe mencionar cualquier tema importante dentro de las estructuras de datos y vincular a los temas relacionados. Dado que la Documentación para estructuras de datos es nueva, es posible que deba crear versiones iniciales de esos temas relacionados.

Examples

Introducción a las estructuras de datos

Una estructura de datos es una forma de organizar y almacenar información.

Deja un "¡Hola mundo!" cadena sea la información que necesitamos para organizar y almacenar en memoria direccionable por byte.

Cada carácter ASCII requiere 7 bits de almacenamiento. La mayoría de los sistemas reservan 8 bits (1 byte) para cada carácter, por lo que cada carácter en "¡Hola, mundo!" se almacena en una unidad individual de memoria del tamaño de un byte, una tras otra, consecutivamente.

Necesitamos una única referencia a nuestra cadena a pesar de que abarca varias direcciones de memoria, por lo que usamos la dirección del primer carácter de la cadena, 'H'. Se puede acceder a todos los demás caracteres en *la dirección de 'H' + el índice de ese carácter* utilizando *caracteres de índice* cero.

Queremos imprimir nuestra cadena, "¡Hola mundo!" Sabemos su dirección en la memoria, que suministramos a la función de impresión, pero ¿cómo sabe la función de impresión para detener la impresión de ubicaciones de memoria consecutivas? Un enfoque común es agregar el carácter nulo, '\ 0', a la cadena. Cuando la función de impresión encuentra el carácter nulo, sabe que ha llegado al final de la cadena.

¡Hemos definido una forma de organizar y almacenar nuestra cadena, es decir, una estructura de datos! Esta estructura de datos muy simple es una matriz de caracteres terminada en nulo, que es una forma de organizar y almacenar una cadena.

Array: una estructura de datos simple

> Una estructura de datos de matriz se utiliza para almacenar objetos similares (o valores de datos) en un bloque de memoria contiguo. La estructura de datos de la matriz tiene un tamaño fijo, que determina el número de valores de datos que se

pueden almacenar en ella.

Array: El Camino C ++

En el lenguaje de programación C ++, podemos declarar una matriz estática de la siguiente manera

```
int arrayName[100];
```

Aquí hemos declarado una matriz llamada "arrayName" que puede almacenar hasta 100 valores, todos los cuales son del mismo tipo, es decir, un entero.

Ahora, vamos a discutir algunas ventajas y desventajas de esta estructura de datos

1. Podemos acceder a los valores de datos almacenados en Array en tiempo constante, es decir, la complejidad del tiempo es O (1) . Por lo tanto, si queremos acceder al valor de los datos almacenados en la i-ésima posición, no necesitamos comenzar desde la posición inicial y avanzar a la i-ésima posición, sino que podemos saltar directamente a la i-ésima posición, lo que ahorra tiempo de computación.
2. Insertar un elemento en medio de una matriz no es una tarea eficiente. Supongamos que queremos agregar un nuevo elemento en la matriz en la posición i-th, luego necesitamos mover todo el elemento en la posición (i-th) y (i + 1 th) para crear espacio para el nuevo elemento. Ejemplo: `1 4 2 0` es una matriz con 4 elementos, ahora queremos insertar 3 en la 2ª posición y luego necesitamos mover 4,2 y 0 una posición más para crear espacio para 3.

```
1 3 4 2 0
```

3. Similar a la inserción del elemento, la eliminación de un elemento de una i-th posición en una matriz tampoco es eficiente, ya que necesitamos mover todos los elementos delante del elemento eliminado en 1 bloque para llenar el espacio vacío creado por el elemento eliminado. elemento.

Estas son 3 características simples de una matriz. Aquí puede creer que la matriz no es una estructura de datos eficiente, pero en la práctica, la ventaja de una matriz puede superar sus desventajas. Esto depende en gran medida del tipo de propósito que desee cumplir, es posible que no desee insertar o eliminar elementos con la frecuencia que desee para acceder a ellos; en ese caso, una matriz es una estructura de datos absolutamente perfecta.

El único propósito de introducir esta estructura de datos es asegurarse de que simplemente no elija la Estructura de datos en función del número de ventajas y desventajas, pero siempre debe intentar analizar la importancia de la Estructura de datos teniendo en cuenta su problema, por ejemplo, Si va a pasar mucho tiempo accediendo a los valores de los datos en comparación con su inserción o eliminación, en ese caso, tenemos que dar más peso a la ventaja sobre la desventaja.

Lea Empezando con estructuras de datos en línea: https://riptutorial.com/es/data-structures/topic/2378/empezando-con-estructuras-de-datos

Capítulo 2: Apilar

Examples

Introducción a la pila

La pila es una estructura de datos LIFO (último en entrar, primero en salir), es decir, el elemento más reciente (o "último en") agregado a la pila será el primer elemento eliminado ("primero en salir").

Consideremos el ejemplo de los libros en una caja. Solo se puede agregar o quitar un libro de la caja a la vez, y solo se puede agregar y quitar desde la parte superior.

Ahora, la caja con los dos primeros libros se ve así:

```
|--------|
| book 2 | ◀──── top of box
|--------|
| book 1 | ◀──── bottom of box
|--------|
```

Si añadimos el libro 3, estará en la parte superior. El resto de los libros, que se agregaron antes del libro 3, permanecerán debajo:

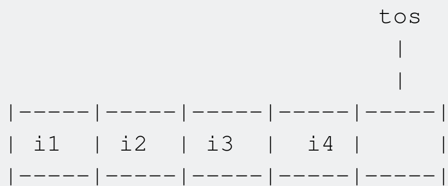
```
|--------|
| book 3 | ◀──── top of box
|--------|
| book 2 |
|--------|
| book 1 | ◀──── bottom of box
|--------|
```

La caja es como una pila y los libros son elementos de datos. Agregar un libro al cuadro es la operación de empuje, mientras que quitar / obtener un libro de la parte superior del cuadro es la operación emergente. Si realizamos la operación pop, obtendremos el libro 3, y la caja volverá a ser como era antes de que presionáramos el libro 3. Esto significa que el último elemento (o el más reciente) que pusimos fue el primer elemento que salió (LIFO). Para obtener el libro 1, tenemos que realizar dos pops más: uno para eliminar el libro 2 y el segundo para obtener el libro 1.

Implementación de la pila utilizando una matriz. Para esto, necesitamos un índice que apunte a la ubicación superior (tos). Cada vez que se inserta un elemento en la pila, los incrementos de tos en uno y cada vez que se realiza una operación de apertura, el índice que apunta a la parte superior de la pila (tos) se reduce en uno.

EMPUJAR:

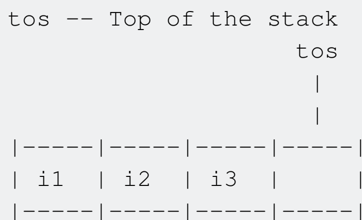
Antes de la operación PUSH



```
                              tos
                               |
                               |
        |-----|-----|-----|-----|-----|
        | i1  | i2  | i3  |  i4 |     |
        |-----|-----|-----|-----|-----|
```

```
void push(dataElment item){
    stack[top]=item; //item = i4
    top++;

}
```

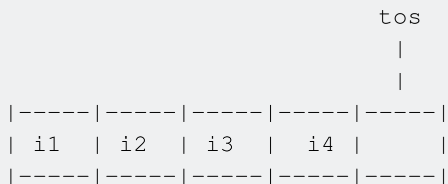
Después de la operación PUSH La pila tiene un puntero a la parte superior de la pila. Cada vez que se llama a la operación de inserción, coloca el valor en la parte superior y lo actualiza.



```
        tos -- Top of the stack
                        tos
                         |
                         |
        |-----|-----|-----|-----|
        | i1  | i2  | i3  |     |
        |-----|-----|-----|-----|
```

POP: la operación emergente elimina el contenido de la parte superior de la pila y actualiza el valor de tos disminuyendo en 1

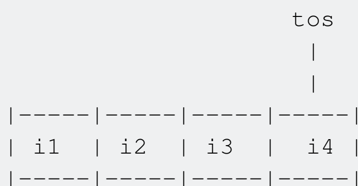
Antes de la operación pop:



```
                              tos
                               |
                               |
        |-----|-----|-----|-----|-----|
        | i1  | i2  | i3  |  i4 |     |
        |-----|-----|-----|-----|-----|
```

```
dataElment pop(){
    dataElment value = stack[tos--];
    return value;
}
```

Después de la operación pop:



```
                        tos
                         |
                         |
        |-----|-----|-----|-----|
        | i1  | i2  | i3  |  i4 |
        |-----|-----|-----|-----|
        When a push operation is performed it overwrites i4.
```

Usando pilas para encontrar palíndromos.

Un palíndromo es una palabra que puede leerse en ambos sentidos, como 'kayak'.

Este ejemplo mostrará cómo encontrar si una palabra dada es un palíndromo utilizando Python3.

Primero, debemos convertir una cadena en una pila (usaremos matrices como pilas).

```
str = "string"
stk = [c for c in str] #this makes an array: ["s", "t", "r", "i", "n", "g"]
stk.append("s") #adds a letter to the array
stk.pop() #pops the last element
```

Ahora, tenemos que invertir la palabra.

```
def invert(str):
    stk = [c for c in str]
    stk2 = []
    while len(stk) > 0:
        stk2.append(stk.pop())
    #now, let's turn stk2 into a string
    str2 = ""
    for c in stk2:
        str2 += c
    return str2
```

Ahora podemos comparar la palabra y la forma invertida.

```
def palindrome(str):
    return str == invert(str)
```

Implementación de la pila utilizando una matriz y una lista enlazada

Implementación de Array

```
#include<stdio.h>

#define MAX 100000                    //Global variables
int top = -1;
int a[MAX];

void push(int x){

    if(top == MAX-1){                         // Check whether stack is full
        printf("Stack Overflow\n");
        return;
    }

    a[++top] = x;                            // Otherwise increment top and insert x
}

void pop(){
    if(top == -1){                                    // if top = -1, empty stack
        printf("Empty stack\n");
        return;
    }
    top--;
```

```
}

void print(){                         //printing stack values

    for(int i=0;i <= top;i++){
        printf("%d ",a[i]);
    }
    printf("\n");
}

void topValue(){                        // Method can print the top value of a stack
    printf("%d",a[top]);
}
int main(){

    push(5);
    push(20);
    push(15);
    print();
    pop();
    print();
    push(35);
    print();
    topValue();

}
```

Implementación de listas enlazadas

```
#include<stdio.h>
#include<malloc.h>

 struct node{
    int data;
    node* link;
};
node* top = NULL;

void push(int data){

    node* temp = (struct node*)malloc(sizeof(struct node*));
    temp->data = data;
    temp->link = top;
    top = temp;
}

void pop(){

    if(top == NULL) return;
    node* temp = top;
    top = top->link;
    free(temp);
}
void print(){

    node* t = top;
    while(t != NULL){
        printf("%d ",t->data);
        t=t->link;
    }
```

```
    printf("\n");
}

void topValue(){
    printf("%d ",top->data);
}
int main(){

    push(5);
    push(20);
    push(15);
    print();
    pop();
    print();
    push(35);
    print();
    topValue();
}
```

Comprobación de paréntesis equilibrados

Se considera que un corchete es uno de los siguientes caracteres: `(` , `)` , `{` , `}` , `[` o `]` .

Se considera que dos corchetes son un par emparejado si el paréntesis de apertura (es decir, `(` , `[` , o `{`) aparece a la izquierda de un corchete de cierre (es decir, `)` , `]` o `}`) del mismo tipo. Hay tres tipos de pares coincidentes de paréntesis: `[]` , `{}` y `()` .

Un par de paréntesis coincidentes no se equilibra si el conjunto de paréntesis que encierra no se empareja. Por ejemplo, `{[(])}` no está equilibrado porque los contenidos entre `{` y `}` no están equilibrados. El par de corchetes encierra un corchete de apertura único, desequilibrado, `(` , y el par de paréntesis encierra un corchete de cierre simple, desequilibrado, `]` .

Por esta lógica, decimos que una secuencia de corchetes se considera equilibrada si se cumplen las siguientes condiciones:

> No contiene paréntesis inigualables.
>
> El subconjunto de corchetes encerrados dentro de los límites de un par de corchetes coincidentes también es un par de corchetes coincidentes.

Algoritmo:

1. Declara una pila (por ejemplo, `stack`).
2. Ahora atraviesa la cadena de entrada.

- a) Si el carácter actual es un corchete inicial (es decir, `(` o `{` o `[`), luego empújelo para apilarlo).
- b) Si el carácter actual es un corchete de cierre (es decir `)` o `}` o `]`), salte de la pila. Si el carácter resaltado es el corchete inicial coincidente, los paréntesis están bien, de lo contrario `not balanced` están `not balanced` .
- c) Si el carácter actual es un corchete de cierre (es decir `)` o `}` o `]`) y la pila está vacía, entonces el paréntesis `not balanced` está `not balanced` .

3. Después de completar el recorrido, si queda algún corchete de inicio en la pila, la cadena `not balanced` estará `not balanced` contrario tendremos una cadena `balanced` .

Lea Apilar en línea: https://riptutorial.com/es/data-structures/topic/6979/apilar

Capítulo 3: Árbol de búsqueda binaria

Examples

Creando un nodo en BST

El árbol de búsqueda binaria (BST) es una estructura de datos jerárquica con un solo puntero al nodo raíz.

El Nodo en el BST generalmente contiene "elementos" (como números o nombres) para una búsqueda rápida. Cada nodo tiene a lo sumo dos hijos (izquierdo y derecho). Cada nodo está organizado por algún campo de datos clave. Para cada nodo en BST, su clave es mayor que la clave izquierda del niño y menor que la clave derecha del niño

Una estructura típica de nodo (que almacena un número entero) sería

```
struct bst_node {
    int item;
    bst_node* left;
    bst_node* right;
};
```

Solo habrá un nodo raíz de BST. El nodo raíz puede ser creado por

```
bst_node* root = NULL;
root = (bst_node*) malloc(sizeof(bst_node));
```

Para establecer la clave del elemento de la raíz a 10.

```
root->item = 10;
```

insertando un nodo en el árbol de búsqueda binario

```
struct tree{
    int a;
    tree* right;
    tree* left;
};
tree* root=NULL;
void insert(tree*& in, int b){
        if(in){
            if(in->a<b)
                    insert(in->right,b);
            else if(in->a>b)
                    insert(in->left,b);
            else
                cout<<"the value is already in the tree."<<endl;
        }else{
            tree* temp = new tree;
            temp->a=b;
```

```
            temp->right=NULL;
            temp->left=NULL;
            in=temp;
        }
}
```

Lea Árbol de búsqueda binaria en línea: https://riptutorial.com/es/data-structures/topic/6161/arbol-de-busqueda-binaria

Capítulo 4: Árbol de segmentos

Examples

Introducción al árbol de segmentos

Supongamos que tenemos una matriz:

```
+-------+-----+-----+-----+-----+-----+-----+
| Index |  0  |  1  |  2  |  3  |  4  |  5  |
+-------+-----+-----+-----+-----+-----+-----+
| Value |  -1 |  3  |  4  |  0  |  2  |  1  |
+-------+-----+-----+-----+-----+-----+-----+
```

Queremos realizar alguna consulta en esta matriz. Por ejemplo:

- ¿Cuál es el mínimo de índice **2** a índice **4** ? -> 0
- ¿Cuál es el máximo de índice **0** a índice **3** ? -> 4
- ¿Cuál es la suma del índice **1** al índice **5** ? -> 10

¿Cómo lo averiguamos?

Fuerza bruta:
Podríamos atravesar la matriz desde el índice inicial hasta el índice final y responder nuestra consulta. En este enfoque, cada consulta toma tiempo `O(n)` donde **n** es la diferencia entre el índice de inicio y el índice de finalización. Pero ¿y si hay millones de números y millones de consultas? Para **m** consultas, tomaría `O(mn)` tiempo. Por lo tanto, para valores de 10^5 en nuestra matriz, si realizamos consultas de 10^5, en el peor de los casos, tendremos que atravesar elementos de 10^{10}.

Programación dinámica:
Podemos crear una matriz de 6X6 para almacenar los valores para diferentes rangos. Para la consulta de rango mínimo (RMQ), nuestra matriz se vería así:

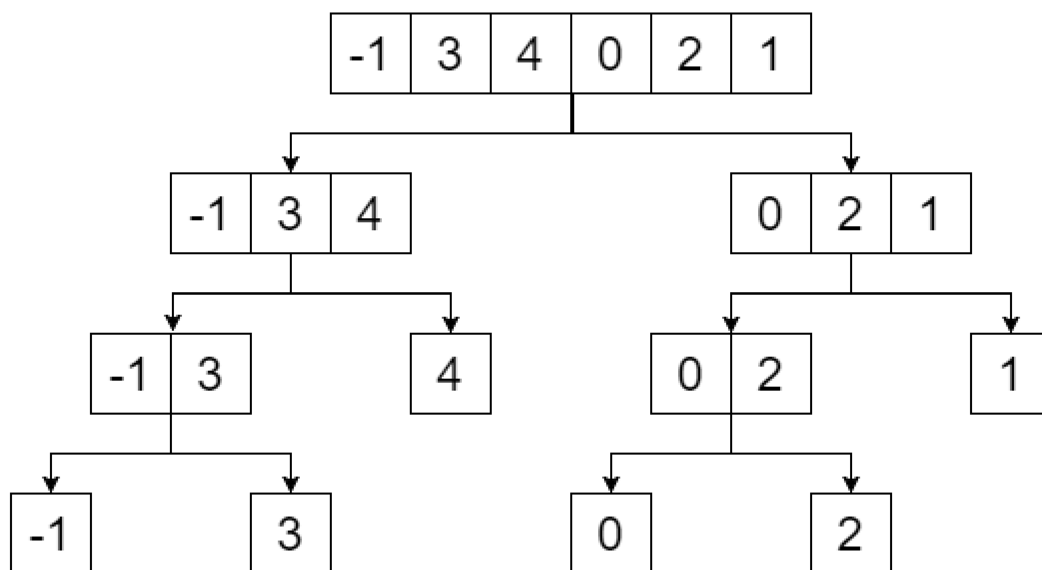
```
              0     1     2     3     4     5
     +-----+-----+-----+-----+-----+-----+-----+
     |     |  -1 |  3  |  4  |  0  |  2  |  1  |
     +-----+-----+-----+-----+-----+-----+-----+
  0  |  -1 |  -1 |  -1 |  -1 |  -1 |  -1 |  -1 |
     +-----+-----+-----+-----+-----+-----+-----+
  1  |  3  |     |  3  |  3  |  0  |  0  |  0  |
     +-----+-----+-----+-----+-----+-----+-----+
  2  |  4  |     |     |  4  |  0  |  0  |  0  |
     +-----+-----+-----+-----+-----+-----+-----+
  3  |  0  |     |     |     |  0  |  0  |  0  |
     +-----+-----+-----+-----+-----+-----+-----+
  4  |  2  |     |     |     |     |  2  |  1  |
     +-----+-----+-----+-----+-----+-----+-----+
  5  |  1  |     |     |     |     |     |  1  |
     +-----+-----+-----+-----+-----+-----+-----+
```

Una vez que tengamos esta estructura de matriz, sería suficiente responder a todas las RMQ. Aquí, **Matrix [i] [j]** almacenaría el valor mínimo de index- **i** a index- **j** . Por ejemplo: el mínimo de índice **2** a índice **4** es **Matriz [2] [4]** = **0** . Esta matriz responde a la consulta en tiempo `O(1)` , pero se necesita tiempo **O (n²)** para construir esta matriz y espacio `O(n²)` para almacenarla. Entonces, si **n** es un número realmente grande, esto no se escala muy bien.

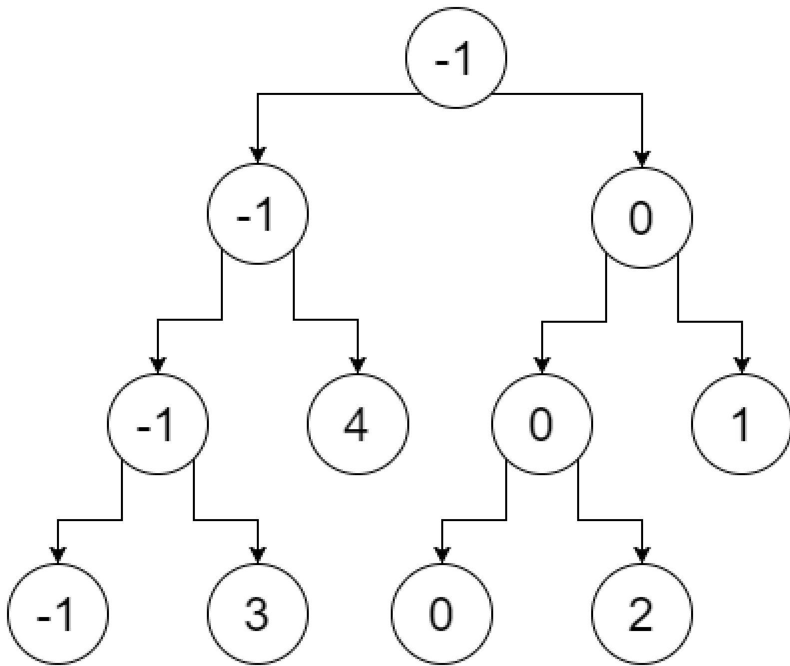
Árbol de segmentos:

Un árbol de segmentos es una estructura de datos de árbol para almacenar intervalos o segmentos. Permite consultar cuál de los segmentos almacenados contiene un punto dado. Se necesita `O(n)` tiempo para construir un árbol de segmentos, se necesita `O(n)` espacio para mantenerlo y responde a una consulta en tiempo `O(logn)` .

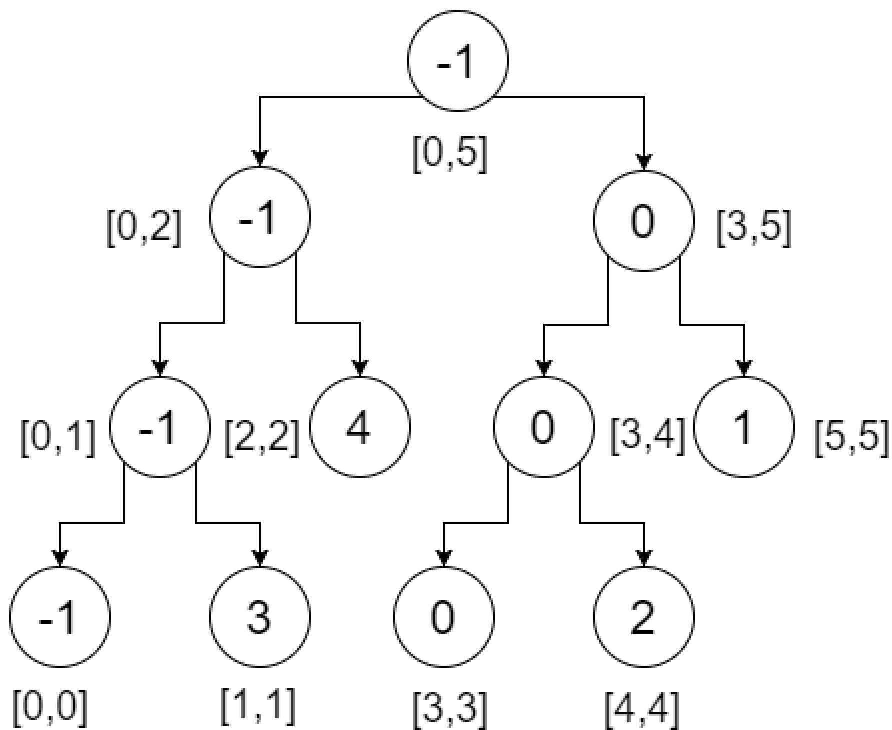
El árbol de segmentos es un árbol binario y los elementos de la matriz dada serán las hojas de ese árbol. Crearemos el árbol de segmentos dividiendo la matriz por la mitad hasta que alcancemos un único elemento. Así que nuestra división nos proporcionaría:



Ahora, si reemplazamos los nodos que no son hojas con el valor mínimo de sus hojas, obtenemos:



Este es nuestro árbol de segmentos. Podemos observar que el nodo raíz contiene el valor mínimo de toda la matriz (rango [0,5]), su hijo izquierdo contiene el valor mínimo de nuestra matriz izquierda (rango [0,2]), el hijo derecho contiene el mínimo valor de nuestra matriz derecha (rango [3,5]) y así sucesivamente. Los nodos de la hoja contienen el valor mínimo de cada punto individual. Obtenemos:



Ahora vamos a hacer una consulta de rango en este árbol. Para hacer una consulta de rango, necesitamos verificar tres condiciones:

- Superposición parcial: comprobamos ambas hojas.
- Superposición total: Devolvemos el valor almacenado en el nodo.
- Sin superposición: Devolvemos un valor muy grande o infinito.

Digamos que queremos verificar el rango **[2,4]** , eso significa que queremos encontrar el mínimo de índice **2** a **4** . Comenzaremos desde la raíz y comprobaremos si el rango de nuestros nodos está superpuesto por nuestro rango de consulta o no. Aquí,

- **[2,4]** no se superpone completamente **[0,5]** . Así que vamos a revisar ambas direcciones.
  - En el subárbol izquierdo, **[2,4]** se superpone parcialmente **[0,2]** . Revisaremos ambas direcciones.
    - En el subárbol izquierdo, **[2,4]** no se superpone **[0,1]** , por lo que esto no contribuirá a nuestra respuesta. Volvemos al **infinito** .
    - En el subárbol derecho, **[2,4]** se superpone totalmente **[2,2]** . Devolvemos **4** .
      El mínimo de estos dos valores devueltos (4, infinito) es **4** . Obtenemos **4** de esta porción.
  - En el subárbol derecho, **[2,4]** se superpone parcialmente. Así que vamos a revisar ambas direcciones.
    - En el subárbol izquierdo, **[2,4]** se superpone completamente **[3,4]** . Devolvemos **0** .
    - En el subárbol derecho, **[2,4]** no se superpone **[5,5]** . Volvemos al **infinito** .
      El mínimo de estos dos valores devueltos (0, infinito) es **0** . Obtenemos **0** de esta parte.
- El mínimo de los valores devueltos (4,0) es **0** . Este es nuestro mínimo deseado en el rango **[2,4]** .

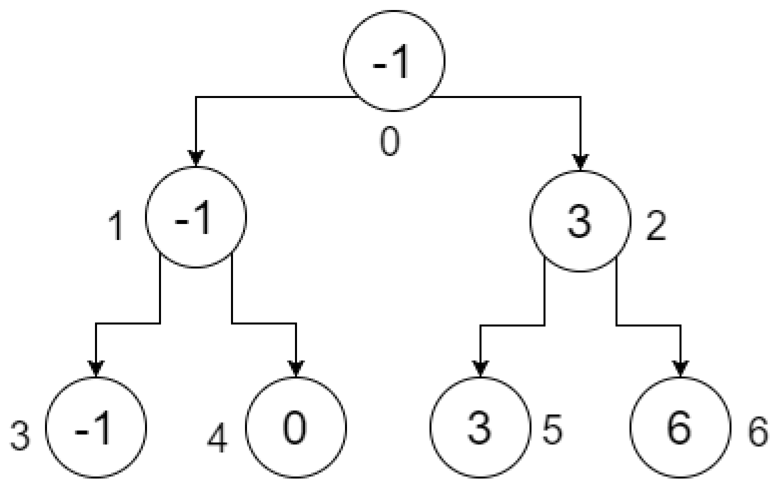
Ahora podemos verificar que, sin importar cuál sea nuestra consulta, se necesitarían un máximo de **4** pasos para encontrar el valor deseado de este árbol de segmentos.

Utilizar:

- Intervalo mínimo de consulta
- El ancestro común más bajo
- Propagación perezosa
- Suma de Subárbol Dinámico
- Negligencia y min
- Campos duales
- Encontrar k-th elemento más pequeño
- Encontrar el número de pares desordenados

Implementación de Segment Tree Using Array

Digamos que tenemos una matriz: `Item = {-1, 0, 3, 6}` . Queremos construir **una** matriz de **SegmentTree** para averiguar el valor mínimo en un rango determinado. Nuestro árbol de segmentos se verá como:



Los números debajo de los nodos muestran los índices de cada valor que almacenaremos en nuestra matriz **SegmentTree** . Podemos ver que, para almacenar 4 elementos, necesitábamos una matriz de tamaño 7. Este valor está determinado por:

```
Procedure DetermineArraySize(Item):
multiplier := 1
n := Item.size
while multiplier < n
    multiplier := multiplier * 2
end while
size := (2 * multiplier) - 1
Return size
```

Entonces, si tuviéramos una matriz de longitud 5, el tamaño de nuestra matriz **SegmentTree** sería: (**8** * **2**) - **1** = **15** . Ahora, para determinar la posición del hijo izquierdo y derecho de un nodo, si un nodo está en el índice **i** , entonces la posición de su:

- Niño izquierdo se denota por: **(2** * **i)** + **1** .
- Right Child se denota por: **(2** * **i)** + **2** .

Y el índice del **padre** de cualquier **nodo** en el índice **i** se puede determinar por: **(i** - **1)** / **2** .

Entonces, la matriz **SegmentTree que** representa nuestro ejemplo se vería así:

```
   0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
|  -1 |  -1 |  3  |  -1 |  0  |  3  |  6  |
+-----+-----+-----+-----+-----+-----+-----+
```

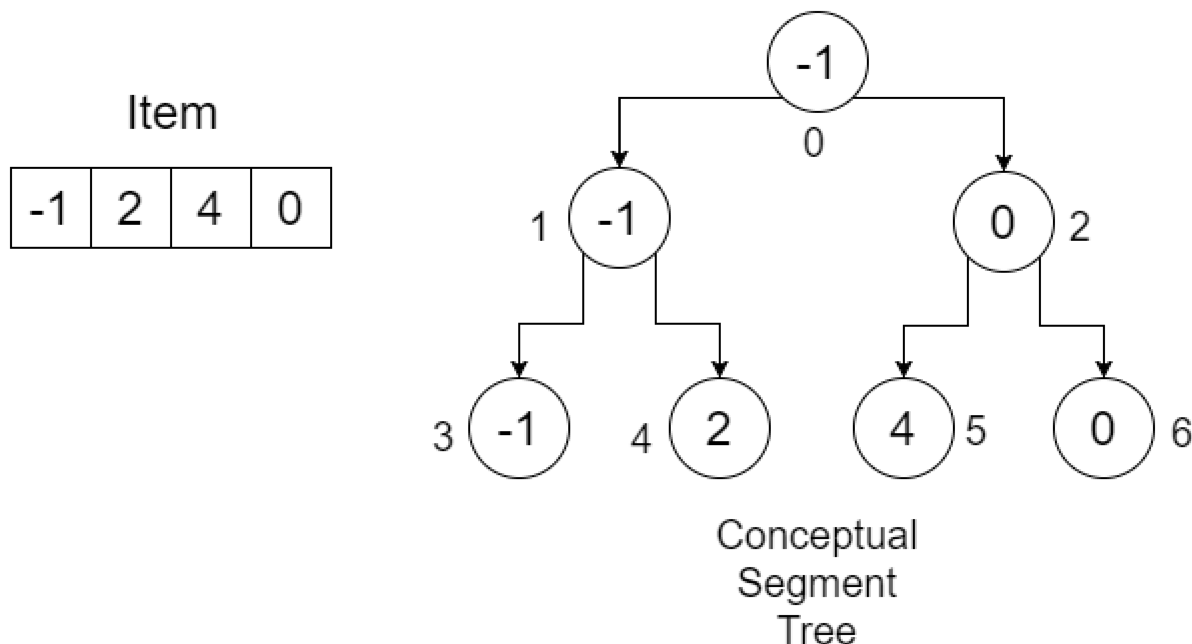
Veamos el pseudo-código para construir esta matriz:

```
Procedure ConstructTree(Item, SegmentTree, low, high, position):
if low is equal to high
    SegmentTree[position] := Item[low]
else
    mid := (low+high)/2
    constructTree(Item, SegmentTree, low, mid, 2*position+1)
    constructTree(Item, SegmentTree, mid+1, high, 2*position+2)
    SegmentTree[position] := min(SegmentTree[2*position+1], SegmentTree[2*position+2])
end if
```

Al principio, tomamos la entrada de los valores e inicializamos la matriz **SegmentTree** con `infinity` utilizando la longitud de la matriz del **elemento** como su tamaño. Llamamos al procedimiento utilizando:

- bajo = Índice inicial de la matriz de **elementos** .
- alto = índice de acabado de la matriz de **elementos** .
- position = 0, indica la **raíz** de nuestro árbol de segmentos.

Ahora, tratemos de entender el procedimiento usando un ejemplo:



El tamaño de nuestra matriz de **elementos** es **4** . Creamos una matriz de longitud **(4** * **2)** - **1** = **7** y las inicializamos con `infinity` . Puede utilizar un valor muy grande para ello. Nuestra matriz se vería como:

```
   0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
| inf | inf | inf | inf | inf | inf | inf |
+-----+-----+-----+-----+-----+-----+-----+
```

Dado que este es un procedimiento recursivo, veremos el funcionamiento de `ConstructTree` utilizando una tabla de recursión que realiza un seguimiento de `low` , `high` , `position` , `mid` y `calling line` en cada llamada. Al principio, llamamos a **ConstructTree (Item, SegmentTree, 0, 3, 0)** . Aquí, `low` no es lo mismo que `high` , obtendremos un `mid` . La `calling line` que `calling line` indica a qué `ConstructTree` se llama después de esta declaración. Denotamos las llamadas de `ConstructTree` dentro del procedimiento como **1** y **2** respectivamente. Nuestra mesa se verá como:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
+-----+------+----------+-----+--------------+
```

Entonces, cuando llamamos a `ConstructTree-1` , pasamos: `low = 0` , `high = mid = 1` , `position =`

`2*position+1 = 2*0+1 = 1` . Una cosa que puedes notar, que es `2*position+1` es el hijo izquierdo de **root** , que es **1** . Dado que `low` no es igual a `high` , obtenemos un valor `mid` . Nuestra mesa se verá como:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
+-----+------+----------+-----+--------------+
|  0  |   1  |     1    |  0  |       1      |
+-----+------+----------+-----+--------------+
```

En la siguiente llamada recursiva, pasamos `low = 0` , `high = mid = 0` , `position = 2*position+1 = 2*1+1=3` . Nuevamente el hijo izquierdo del índice **1** es **3** . Aquí, `low` es e `high` , así que configuramos `SegmentTree[position] = SegmentTree[3] = Item[low] = Item[0] = -1` . Nuestra matriz **SegmentTree** se verá como:

```
   0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
| inf | inf | inf |  -1 | inf | inf | inf |
+-----+-----+-----+-----+-----+-----+-----+
```

Nuestra tabla de recursión se verá así:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
+-----+------+----------+-----+--------------+
|  0  |   1  |     1    |  0  |       1      |
+-----+------+----------+-----+--------------+
|  0  |   0  |     3    |     |              |
+-----+------+----------+-----+--------------+
```

Como pueden ver, **-1** tiene su posición correcta. Como esta llamada recursiva está completa, volvemos a la fila anterior de nuestra tabla de recursiones. La mesa:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
+-----+------+----------+-----+--------------+
|  0  |   1  |     1    |  0  |       1      |
+-----+------+----------+-----+--------------+
```

En nuestro procedimiento, ejecutamos la llamada a `ConstructTree-2` . Esta vez, pasamos `low = mid+1 = 1` , `high = 1` , `position = 2*position+2 = 2*1+2 = 4` . Nuestra `calling line` cambia a **2** . Obtenemos:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
```

```
+-----+------+----------+-----+--------------+
|  0  |   1  |     1    |  0  |       2      |
+-----+------+----------+-----+--------------+
```

Dado que, `low` es igual a `high` , establecemos: `SegmentTree[position] = SegmentTree[4] = Item[low] = Item[1] = 2` . Nuestra matriz de **SegmentTree** :

```
   0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
| inf | inf | inf |  -1 |  2  | inf | inf |
+-----+-----+-----+-----+-----+-----+-----+
```

Nuestra tabla de recursión:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
+-----+------+----------+-----+--------------+
|  0  |   1  |     1    |  0  |       2      |
+-----+------+----------+-----+--------------+
|  1  |   1  |     4    |     |              |
+-----+------+----------+-----+--------------+
```

De nuevo se puede ver, **2** tiene su posición correcta. Después de esta llamada recursiva, volvemos a la fila anterior de nuestra tabla de recursiones. Obtenemos:

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       1      |
+-----+------+----------+-----+--------------+
|  0  |   1  |     1    |  0  |       2      |
+-----+------+----------+-----+--------------+
```

`SegmentTree[position] = SegmentTree[1] = min(SegmentTree[2*position+1], SegmentTree[2*position+2]) = min(SegmentTree[3], SegmentTree[4]) = min(-1,2) = -1` la última línea de nuestro procedimiento, `SegmentTree[position] = SegmentTree[1] = min(SegmentTree[2*position+1], SegmentTree[2*position+2]) = min(SegmentTree[3], SegmentTree[4]) = min(-1,2) = -1` . Nuestra matriz de **SegmentTree** :

```
   0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
| inf |  -1 | inf |  -1 |  2  | inf | inf |
+-----+-----+-----+-----+-----+-----+-----+
```

Como esta llamada de recursión está completa, volvemos a la fila anterior de nuestra tabla de recursión y llamamos a `ConstructTree-2` :

```
+-----+------+----------+-----+--------------+
| low | high | position | mid | calling line |
+-----+------+----------+-----+--------------+
|  0  |   3  |     0    |  1  |       2      |
```

```
+-----+------+----------+-----+-------------+
```

Podemos ver que la parte izquierda de nuestro árbol de segmentos está completa. Si continuamos de esta manera, después de completar todo el procedimiento, finalmente obtendremos una matriz de **SegmentTree** completada, que se verá así:

```
   0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
|  -1 |  -1 |  0  |  -1 |  2  |  4  |  0  |
+-----+-----+-----+-----+-----+-----+-----+
```

La complejidad de tiempo y espacio de la construcción de esta matriz de **SegmentTree** es: `O(n)` , donde **n** denota el número de elementos en la matriz de **elementos** . Nuestra matriz **SegmentTree** construida se puede usar para realizar *consultas de rango mínimo (RMQ)* . Para construir una matriz para realizar una *consulta de rango máximo* , necesitamos reemplazar la línea:

```
SegmentTree[position] := min(SegmentTree[2*position+1], SegmentTree[2*position+2])
```

con:

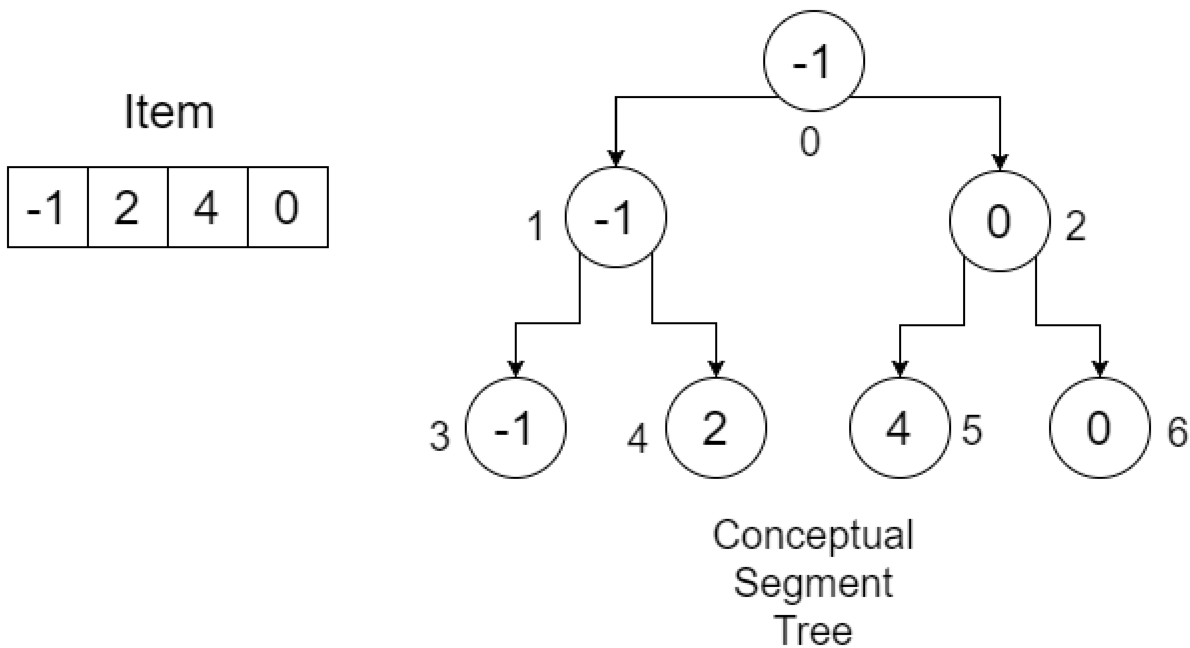
```
SegmentTree[position] := max(SegmentTree[2*position+1], SegmentTree[2*position+2])
```

Realizar una consulta de rango mínimo

El procedimiento para realizar una RMQ ya se muestra en la introducción. El pseudo-código para verificar el rango mínimo de consulta será:

```
Procedure RangeMinimumQuery(SegmentTree, qLow, qHigh, low, high, position):
if qLow <= low and qHigh >= high        //Total Overlap
    Return SegmentTree[position]
else if qLow > high || qHigh < low      //No Overlap
    Return infinity
else                                    //Partial Overlap
    mid := (low+high)/2
    Return min(RangeMinimumQuery(SegmentTree, qLow, qHigh, low, mid, 2*position+1),
               RangeMinimumQuery(SegmentTree, qLow, qHigh, mid+1, high, 2*position+2))
end if
```

Aquí, `qLow = The lower range of our query` , `qHigh = The upper range of our query` . `low = starting index of Item array` , `high = Finishing index of Item array` , `position = root = 0` . Ahora tratemos de entender el procedimiento usando el ejemplo que creamos antes:



Nuestra matriz de **SegmentTree** :

```
   0     1     2     3     4     5     6
+-----+-----+-----+-----+-----+-----+-----+
|  -1 |  -1 |  0  |  -1 |  2  |  4  |  0  |
+-----+-----+-----+-----+-----+-----+-----+
```

Queremos encontrar el mínimo en el rango **[1,3]** .
Dado que este es un procedimiento recursivo, veremos el funcionamiento de `RangeMinimumQuery` utilizando una tabla de recursión que realiza un seguimiento de `qLow` , `qHigh` , `low` , `high` , `position` , `mid` y `calling line` . Al principio, llamamos a **RangeMinimumQuery (SegmentTree, 1, 3, 0, 3, 0.** Aquí, no se cumplen las dos primeras condiciones (superposición parcial). `RangeMinimumQuery` un valor `mid` . La `calling line` indica a qué `RangeMinimumQuery` se llama después de esto declaración. Denotamos las llamadas `RangeMinimumQuery` dentro del procedimiento como **1** y **2** respectivamente. Nuestra tabla se verá así:

```
+------+-------+-----+------+----------+-----+--------------+
| qLow | qHigh | low | high | position | mid | calling line |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   3  |     0    |  1  |       1      |
+------+-------+-----+------+----------+-----+--------------+
```

Así que cuando llamamos `RangeMinimumQuery-1` , pasamos: `low = 0` , `high = mid = 1` , `position = 2*position+1 = 1` . Una cosa que puedes notar, es que `2*position+1` es el hijo izquierdo de un **nodo** . Eso significa que estamos comprobando el hijo izquierdo de **root** . Dado que **[1,3]** se superpone parcialmente **[0,1]** , las dos primeras condiciones no se cumplen, obtenemos una `mid` . Nuestra mesa:

```
+------+-------+-----+------+----------+-----+--------------+
| qLow | qHigh | low | high | position | mid | calling line |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   3  |     0    |  1  |       1      |
+------+-------+-----+------+----------+-----+--------------+
```

```
|   1  |   3   |  0  |   1  |     1    |  0  |      1       |
+------+-------+-----+------+----------+-----+--------------+
```

En la siguiente llamada recursiva, pasamos `low = 0` , `high = mid = 0` , `position = 2*position+1 = 3` . Llegamos a la hoja más a la izquierda de nuestro árbol. Como **[1,3]** no se superpone con **[0,0]** , devolvemos el `infinity` a nuestra función de llamada. Tabla de recursión:

```
+------+-------+-----+------+----------+-----+--------------+
| qLow | qHigh | low | high | position | mid | calling line |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   3  |     0    |  1  |      1       |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   1  |     1    |  0  |      1       |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   0  |     3    |     |              |
+------+-------+-----+------+----------+-----+--------------+
```

Como esta llamada recursiva está completa, volvemos a la fila anterior de nuestra tabla de recursiones. Obtenemos:

```
+------+-------+-----+------+----------+-----+--------------+
| qLow | qHigh | low | high | position | mid | calling line |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   3  |     0    |  1  |      1       |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   1  |     1    |  0  |      1       |
+------+-------+-----+------+----------+-----+--------------+
```

En nuestro procedimiento, ejecutamos la llamada `RangeMinimumQuery-2` . Esta vez, pasamos `low = mid+1 = 1` , `high = 1` y `position = 2*position+2 = 4` . Nuestra `calling line changes to **2**` . Obtenemos:

```
+------+-------+-----+------+----------+-----+--------------+
| qLow | qHigh | low | high | position | mid | calling line |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   3  |     0    |  1  |      1       |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   1  |     1    |  0  |      2       |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  1  |   1  |     4    |     |              |
+------+-------+-----+------+----------+-----+--------------+
```

Así que vamos al hijo correcto del nodo anterior. Esta vez hay una superposición total. `SegmentTree[position] = SegmentTree[4] = 2` el valor `SegmentTree[position] = SegmentTree[4] = 2` .

```
+------+-------+-----+------+----------+-----+--------------+
| qLow | qHigh | low | high | position | mid | calling line |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   3  |     0    |  1  |      1       |
+------+-------+-----+------+----------+-----+--------------+
```

De vuelta en la función de llamada, estamos comprobando cuál es el mínimo de los dos valores devueltos de dos funciones de llamada. Obviamente el mínimo es **2** , devolvemos **2** a la función

de llamada. Nuestra tabla de recursión se ve como:

```
+------+-------+-----+------+----------+-----+--------------+
| qLow | qHigh | low | high | position | mid | calling line |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   3  |     0    |  1  |       1      |
+------+-------+-----+------+----------+-----+--------------+
```

Hemos terminado de mirar la parte izquierda de nuestro árbol de segmentos. Ahora llamaremos a `RangeMinimumQuery-2` desde aquí. Pasaremos `low = mid+1 = 1+1 =2` , `high = 3` y `position = 2*position+2 = 2` . Nuestra mesa:

```
+------+-------+-----+------+----------+-----+--------------+
| qLow | qHigh | low | high | position | mid | calling line |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  0  |   3  |     0    |  1  |       1      |
+------+-------+-----+------+----------+-----+--------------+
|   1  |   3   |  2  |   3  |     2    |     |              |
+------+-------+-----+------+----------+-----+--------------+
```

Hay una superposición total. Entonces devolvemos el valor: `SegmentTree[position] = SegmentTree[2] = 0` . Regresamos a la recursión desde donde se llamó a estos dos niños y obtenemos el mínimo de **(4,0)** , que es **0** y devolvemos el valor.

Después de ejecutar el procedimiento, obtenemos **0** , que es el mínimo de índice **1** a índice **3** .

La complejidad del tiempo de ejecución para este procedimiento es `O(logn)` donde **n** es el número de elementos en la matriz de **elementos** . Para realizar una consulta de rango máximo, necesitamos reemplazar la línea:

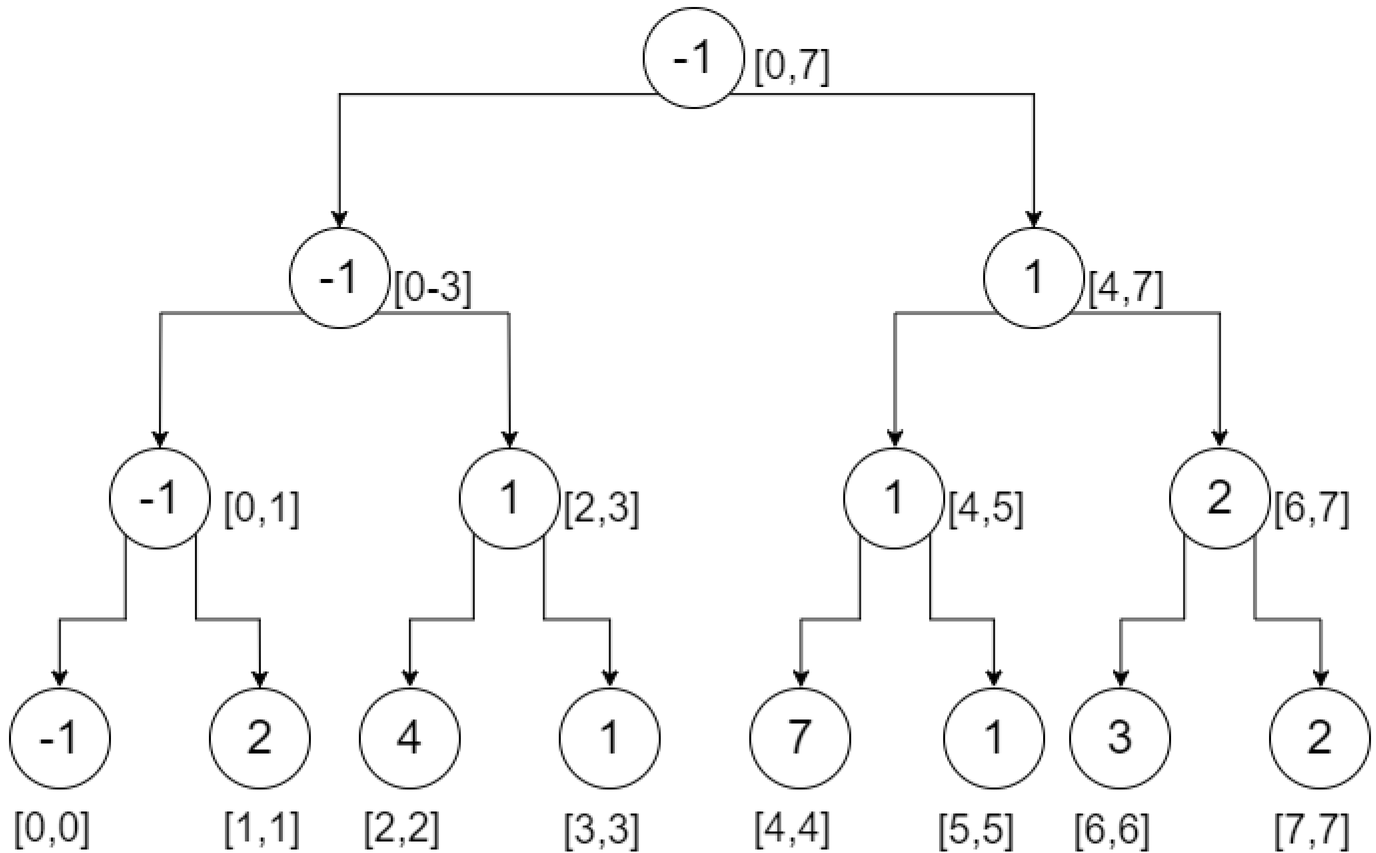
```
Return min(RangeMinimumQuery(SegmentTree, qLow, qHigh, low, mid, 2*position+1),
               RangeMinimumQuery(SegmentTree, qLow, qHigh, mid+1, high, 2*position+2))
```

con:

```
Return max(RangeMinimumQuery(SegmentTree, qLow, qHigh, low, mid, 2*position+1),
               RangeMinimumQuery(SegmentTree, qLow, qHigh, mid+1, high, 2*position+2))
```

Propagación perezosa

Digamos que ya ha creado un árbol de segmentos. Es necesario que actualice los valores de la matriz, esto no solo cambiará los nodos de hoja de su árbol de segmentos, sino también el mínimo / máximo en algunos nodos. Veamos esto con un ejemplo:



Este es nuestro árbol de segmentos mínimo de **8** elementos. Para darle un recordatorio rápido, cada nodo representa el valor mínimo del rango mencionado junto a ellos. Digamos que queremos incrementar el valor del primer elemento de nuestra matriz en **3** . ¿Cómo podemos hacer eso? Seguiremos la forma en que realizamos la RMQ. El proceso se vería como:

- Al principio, atravesamos la raíz. **[0,0]** se superpone parcialmente con **[0,7]** , vamos a ambas direcciones.
    - En el subárbol izquierdo, **[0,0]** se superpone parcialmente con **[0,3]** , vamos a ambas direcciones.
        - En el subárbol izquierdo, **[0,0]** se superpone parcialmente con **[0,1]** , vamos a ambas direcciones.
            - En el subárbol izquierdo, **[0,0]** se superpone totalmente con **[0,0]** , y dado que es el nodo hoja, actualizamos el nodo aumentando su valor en **3** . Y devuelve el valor **-1** + **3** = **2** .
            - En el subárbol derecho, **[1,1]** no se superpone con **[0,0]** , devolvemos el valor en el nodo (**2**).
              El mínimo de estos dos valores devueltos (**2** , **2**) es **2** , por lo que actualizamos el valor del nodo actual y lo devolvemos.
        - En el subárbol derecho **[2,3]** no se superpone con **[0,0]** , devolvemos el valor del nodo. (**1**).
          Como el mínimo de estos dos valores devueltos (**2** , **1**) es **1** , actualizamos el valor del nodo actual y lo devolvemos.
    - En el subárbol derecho **[4,7]** no se superpone con **[0,0]** , devolvemos el valor del nodo. (**1**).
- Finalmente, el valor del nodo raíz se actualiza ya que el mínimo de los dos valores devueltos **(1,1)** es **1** .

Podemos ver que la actualización de un solo nodo requiere una complejidad de tiempo `O(logn)` , donde **n** es el número de nodos hoja. Entonces, si nos piden que actualicemos algunos nodos de **i** a **j** , necesitaremos complejidad de tiempo `O(nlogn)` . Esto se vuelve engorroso para un gran valor de **n** . Seamos *perezosos* , es decir, trabajemos solo cuando sea necesario. ¿Cómo? Cuando necesitamos actualizar un intervalo, actualizaremos un nodo y marcaremos a su hijo que necesita ser actualizado y lo actualizaremos cuando sea necesario. Para esto necesitamos una matriz **perezosa** del mismo tamaño que la de un árbol de segmentos. Inicialmente, todos los elementos de la matriz **diferida** serán **0, lo** que representa que no hay una actualización pendiente. Si hay un elemento distinto de cero en **perezoso [i]** , este elemento debe actualizar el nodo **i** en el árbol de segmentos antes de realizar cualquier operación de consulta. ¿Cómo vamos a hacer eso? Veamos un ejemplo:

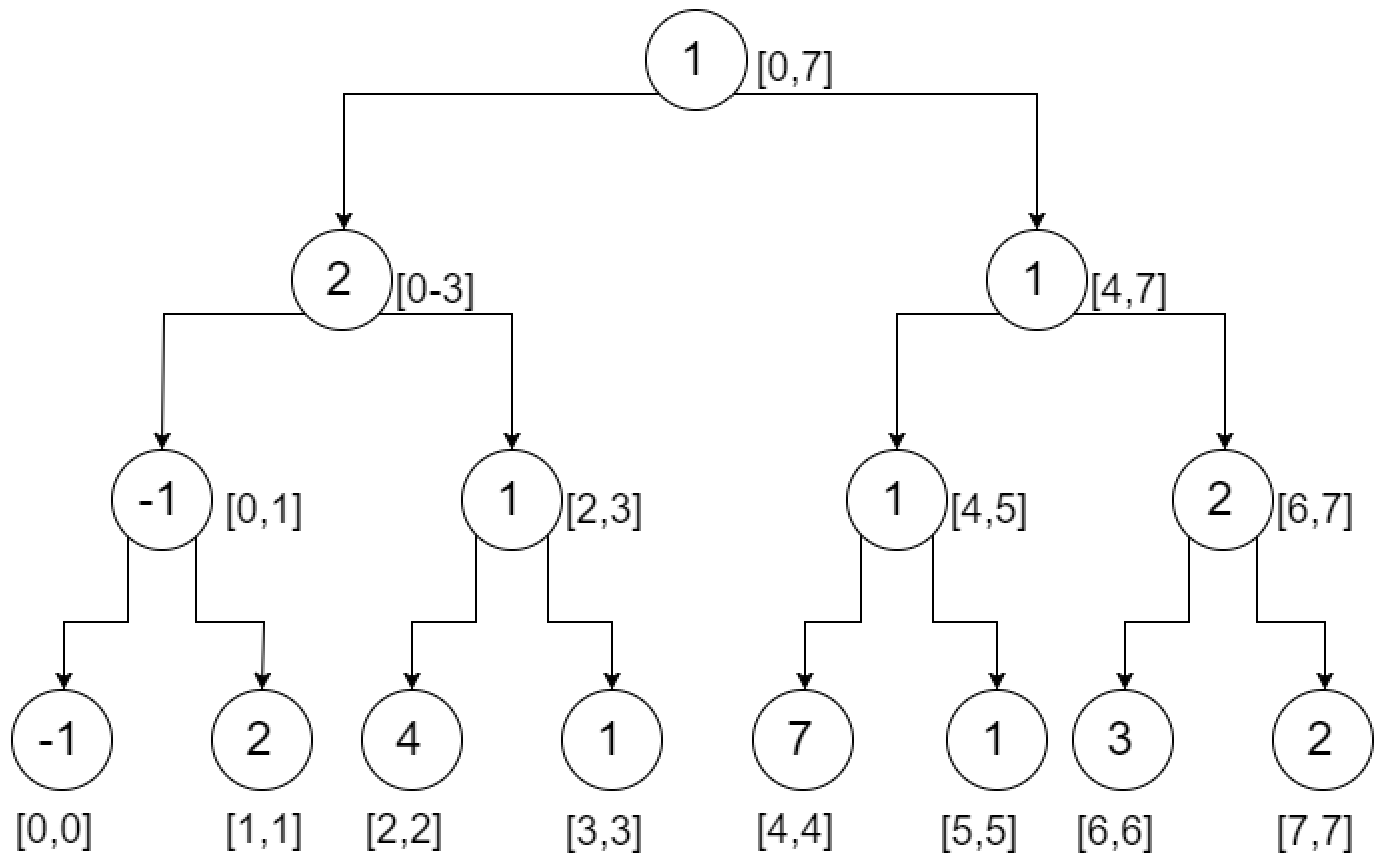
Digamos, para nuestro árbol de ejemplo, queremos ejecutar algunas consultas. Estos son:

- incremento **[0,3]** por **3** .
- incremento **[0,3]** por **1** .
- Incremento **[0,0]** en **2** .

incremento [0,3] por 3:

- Partimos del nodo raíz. Al principio, comprobamos si este valor está actualizado. Para esto verificamos nuestra matriz **perezosa** que es **0** , lo que significa que el valor está actualizado. **[0,3]** se superpone parcialmente **[0,7]** . Así que vamos a las dos direcciones.
  - En el subárbol izquierdo, no hay actualización pendiente. **[0,3]** se superpone totalmente **[0,3]** . Así que actualizamos el valor del nodo por **3** . Entonces el valor se convierte en **-1** + **3** = **2** . Esta vez, no vamos a ir todo el camino. En lugar de bajar, actualizamos el elemento secundario correspondiente en el árbol diferido de nuestro nodo actual y lo incrementamos en **3** . También devolvemos el valor del nodo actual.
  - En el subárbol derecho, no hay actualización pendiente. **[0,3]** no se superpone **[4,7]** . Entonces devolvemos el valor del nodo actual **(1)** .
    El mínimo de dos valores devueltos (**2** , **1**) es **1** . Actualizamos el nodo raíz a **1** .

Nuestro árbol de segmentos y árbol perezoso se vería así:

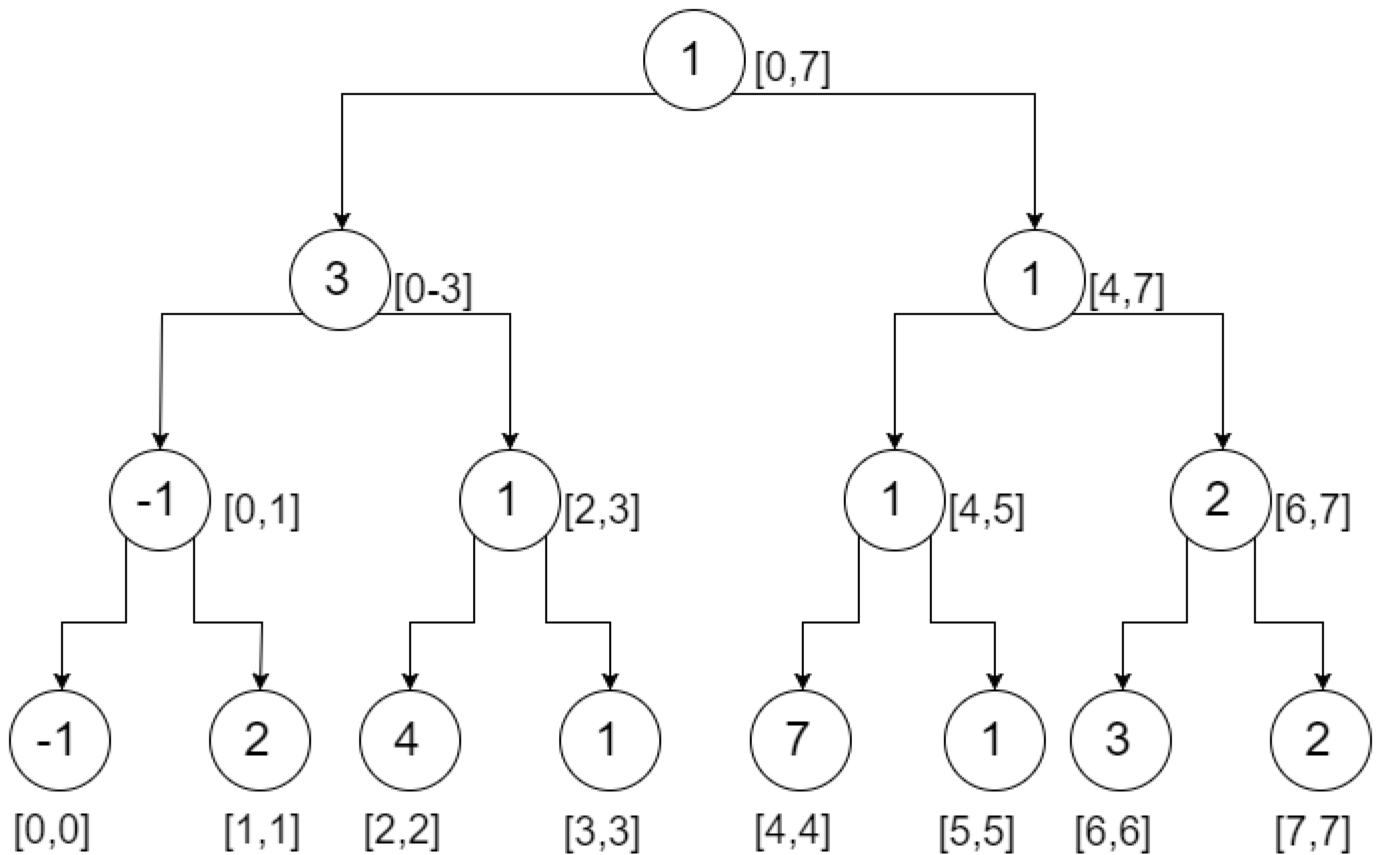


Los valores que no son cero en los nodos de nuestro Árbol perezoso representan, hay actualizaciones pendientes en esos nodos y más abajo. Los actualizaremos si es necesario. Si se nos pregunta, cuál es el mínimo en el rango **[0,3]** , llegaremos al subárbol izquierdo del nodo raíz y, como no hay actualizaciones pendientes, devolveremos **2** , lo cual es correcto. Entonces, este proceso no afecta la corrección de nuestro algoritmo de árbol de segmentos.

incremento [0,3] por 1:

- Partimos del nodo raíz. No hay actualización pendiente. **[0,3]** se superpone parcialmente **[0,7]** . Así que vamos a ambas direcciones.
  - En el subárbol izquierdo, no hay actualización pendiente. **[0,3]** se superpone completamente **[0,3]** . Actualizamos el nodo actual: **2** + **1** = **3** . Dado que este es un nodo interno, actualizamos sus hijos en el Árbol diferido para que se incrementen en **1** . También devolveremos el valor del nodo actual (**3**).
  - En el subárbol derecho, no hay actualización pendiente. **[0,3]** no se superpone **[4,7]** . Devolvemos el valor del nodo actual (**1**).
- Actualizamos el nodo raíz tomando el mínimo de dos valores devueltos (**3** , **1**).

Nuestro árbol de segmentos y árbol perezoso se verá así:



Como puede ver, estamos acumulando las actualizaciones en Lazy Tree pero no lo estamos presionando. Esto es lo que significa la propagación perezosa. Si no lo hubiésemos usado, tuvimos que empujar los valores hasta las hojas, lo que nos costaría una complejidad de tiempo más innecesaria.

incremento [0,0] por 2:

- Partimos del nodo raíz. Como la raíz está actualizada y **[0,0]** se superpone parcialmente **[0,7]** , vamos a ambas direcciones.
    - En el subárbol izquierdo, el nodo actual está actualizado y **[0,0]** se superpone parcialmente **[0,3]** , vamos a ambas direcciones.
        - En el subárbol izquierdo, el nodo actual en Lazy Tree tiene un valor distinto de cero. Así que hay una actualización que aún no se ha propagado a este nodo. Primero vamos a actualizar este nodo en nuestro árbol de segmentos. Entonces esto se convierte en **-1** + **4** = **3** . Luego vamos a propagar este **4** a sus hijos en el árbol perezoso. Como ya hemos actualizado el nodo actual, cambiaremos el valor del nodo actual en Lazy Tree a **0** . Luego **[0,0]** se superpone parcialmente **[0,1]** , por lo que vamos a ambas direcciones.
            - En el nodo izquierdo, el valor debe actualizarse ya que hay un valor distinto de cero en el nodo actual de Lazy Tree. Así que actualizamos el valor a **-1** + **4** = **3** . Ahora, ya que **[0,0]** se superpone totalmente **[0,0]** , actualizamos el valor del nodo actual a **3** + **2** = **5** . Este es un nodo hoja, por lo que no necesitamos propagar más el valor. Actualizamos el nodo correspondiente en Lazy Tree a **0,** ya que todos los valores se han propagado hasta este

nodo. Devolvemos el valor del nodo actual (**5**).

    - En el nodo derecho, el valor debe actualizarse. Entonces el valor se convierte en: **4** + **2** = **6** . Como **[0,0]** no se superpone **[1,1]** , devolvemos el valor del nodo actual (**6**). También actualizamos el valor en Lazy Tree a **0** . No se necesita propagación ya que este es un nodo de hoja.

    Actualizamos el nodo actual utilizando el mínimo de dos valores devueltos (**5** , **6**). Devolvemos el valor del nodo actual (**5**).

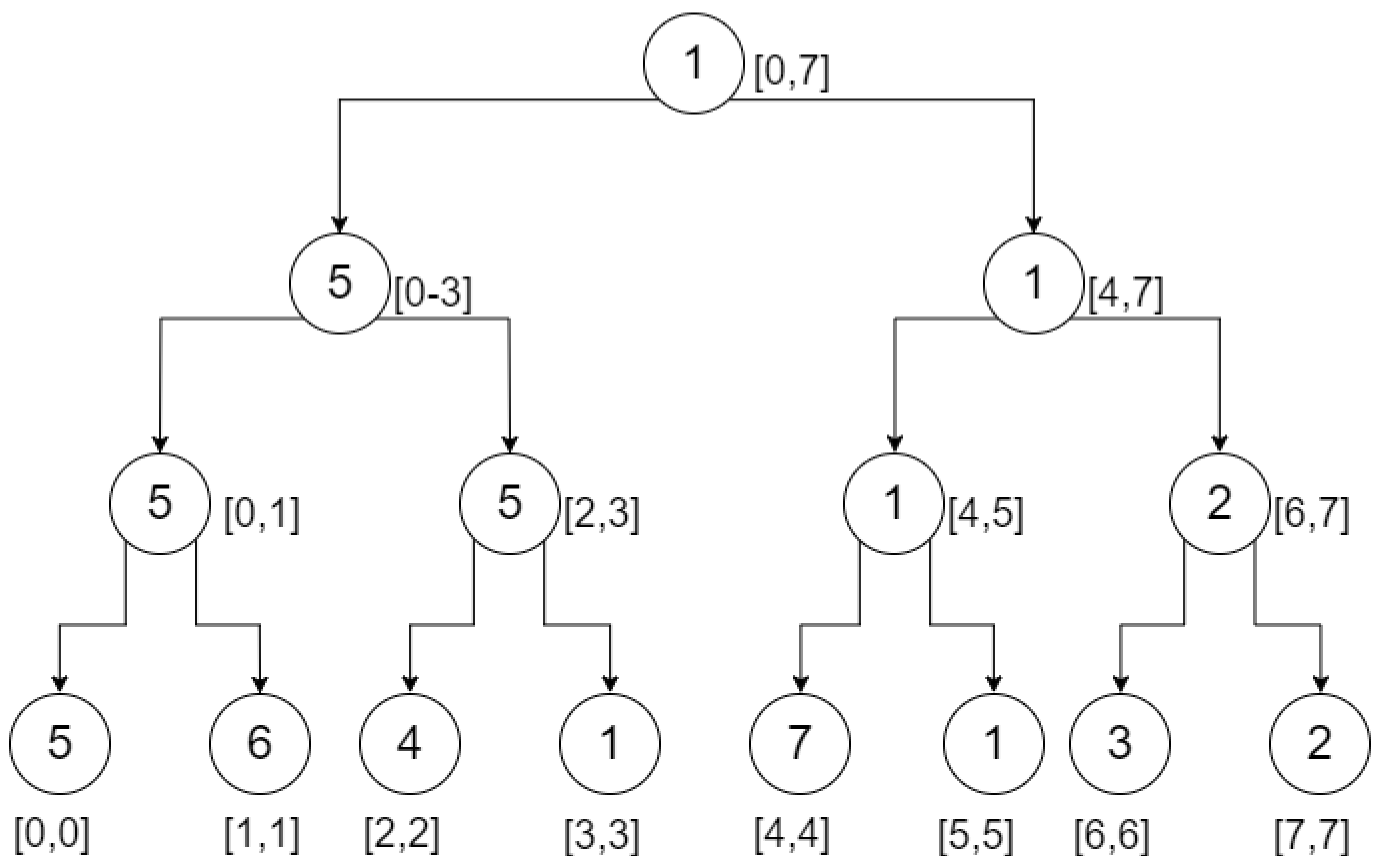
  - En el subárbol derecho, hay una actualización pendiente. Actualizamos el valor del nodo a **1** + **4** = **5** . Como este no es un nodo de hoja, propagamos el valor a sus hijos en nuestro Árbol diferido y actualizamos el nodo actual a **0** . Como **[0,0]** no se superpone con **[2,3]** , devolvemos el valor de nuestro nodo actual (**5**).

  Actualizamos el nodo actual utilizando el mínimo de los valores devueltos (**5** , **5**) y devolvemos el valor (**5**).

- En el subárbol derecho, no hay actualización pendiente y como **[0,0]** no se superpone **[4,7]** , devolvemos el valor del nodo actual (**1**).

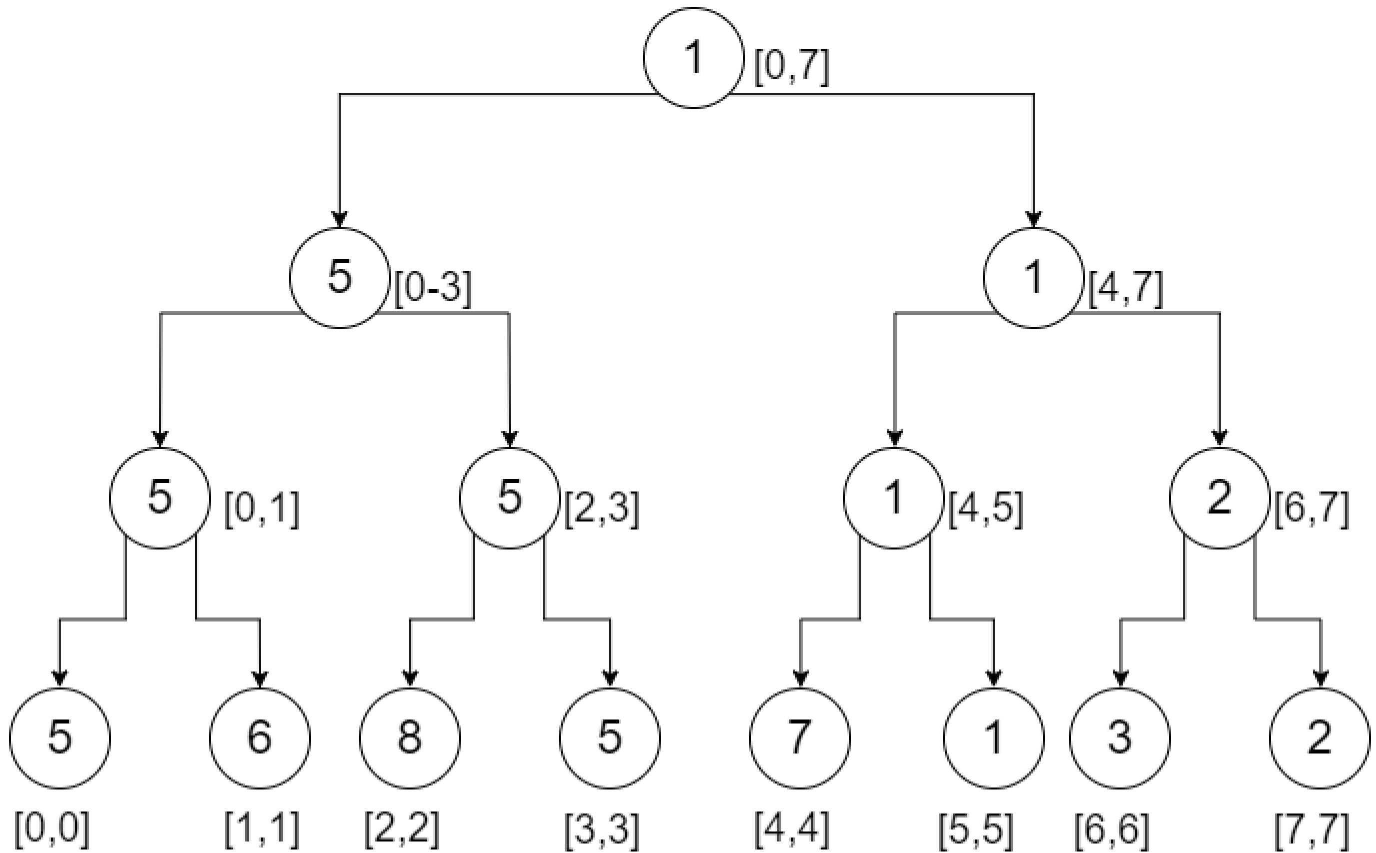
- Actualizamos el nodo raíz utilizando el mínimo de los dos valores devueltos (**5** , **1**).

Nuestro árbol de segmentos y árbol perezoso se verá así:



Podemos notar que, el valor en **[0,0]** , cuando es necesario, obtuvo todo el incremento.

Ahora, si se le pide que encuentre el rango mínimo **[3,5]** , si ha comprendido hasta este punto, puede averiguar fácilmente cómo iría la consulta y el valor devuelto será **1** . Nuestro segmento Tree y Lazy Tree se vería así:



Simplemente hemos seguido el mismo proceso que seguimos para encontrar RMQ con restricciones añadidas de verificación de Lazy Tree en busca de actualizaciones pendientes.

Otra cosa que podemos hacer es en lugar de devolver los valores de cada nodo, ya que sabemos cuál será el nodo secundario de nuestro nodo actual, simplemente podemos verificarlos para encontrar el mínimo de estos dos.

El pseudocódigo para actualizar en Lazy Propagation sería:

```
Procedure UpdateSegmentTreeLazy(segmentTree, LazyTree, startRange,
                                endRange, delta, low, high, position):
if low > high                                                  //out of bounds
    Return
end if
if LazyTree[position] is not equal to 0                        //update needed
    segmentTree[position] := segmentTree[position] + LazyTree[position]
    if low is not equal to high                                //non-leaf node
        LazyTree[2 * position + 1] := LazyTree[2 * position + 1] + delta
        LazyTree[2 * position + 2] := LazyTree[2 * position + 2] + delta
    end if
    LazyTree[position] := 0
end if
if startRange > low or endRange < high                         //doesn't overlap
    Return
end if
if startRange <= low && endRange >= high                       //total overlap
    segmentTree[position] := segmentTree[position] + delta
    if low is not equal to high
```

```
        LazyTree[2 * position + 1] := LazyTree[2 * position + 1] + delta
        LazyTree[2 * position + 2] := LazyTree[2 * position + 2] + delta
    end if
    Return
end if
//if we reach this portion, this means there's a partial overlap
mid := (low + high) / 2
UpdateSegmentTreeLazy(segmentTree, LazyTree, startRange,
                                endRange, delta, low, mid, 2 * position + 1)
UpdateSegmentTreeLazy(segmentTree, LazyTree, startRange,
                                endRange, delta, mid + 1, high, 2 * position + 2)
segmentTree[position] := min(segmentTree[2 * position + 1],
                                                    segmentTree[2 * position + 2])
```

Y el pseudo-código para RMQ en la propagación perezosa será:

```
Procedure RangeMinimumQueryLazy(segmentTree, LazyTree, qLow, qHigh, low, high, position):
if low > high
    Return infinity
end if
if LazyTree[position] is not equal to 0                      //update needed
    segmentTree[position] := segmentTree[position] + LazyTree[position]
    if low is not equal to high
        segmentTree[position] := segmentTree[position] + LazyTree[position]
    if low is not equal to high                              //non-leaf node
        LazyTree[2 * position + 1] := LazyTree[2 * position + 1] + LazyTree[position]
        LazyTree[2 * position + 2] := LazyTree[2 * position + 2] + LazyTree[position]
    end if
    LazyTree[position] := 0
end if
if qLow > high and qHigh < low                                //no overlap
    Return infinity
end if
if qLow <= low and qHigh >= high                              //total overlap
    Return segmentTree[position]
end if
//if we reach this portion, this means there's a partial overlap
mid := (low + high) / 2
Return min(RangeMinimumQueryLazy(segmentTree, LazyTree, qLow, qHigh,
                                        low, mid, 2 * position + 1),
           RangeMinimumQueryLazy(segmentTree, LazyTree, qLow, qHigh,
                                        mid + 1, high, 2 * position + 1)
```

Lea Árbol de segmentos en línea: https://riptutorial.com/es/data-structures/topic/7908/arbol-de-segmentos

Capítulo 5: Cola

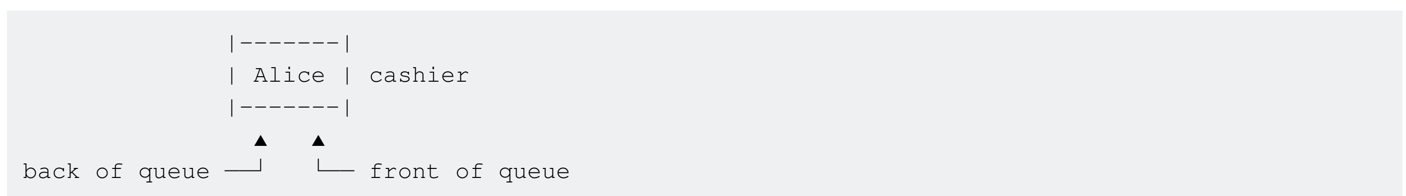
Examples

Introducción a la cola

La cola es una estructura de datos FIFO (primero en entrar, primero en salir), es decir, el primer elemento agregado a la cola será el primer elemento eliminado ("primero en salir").

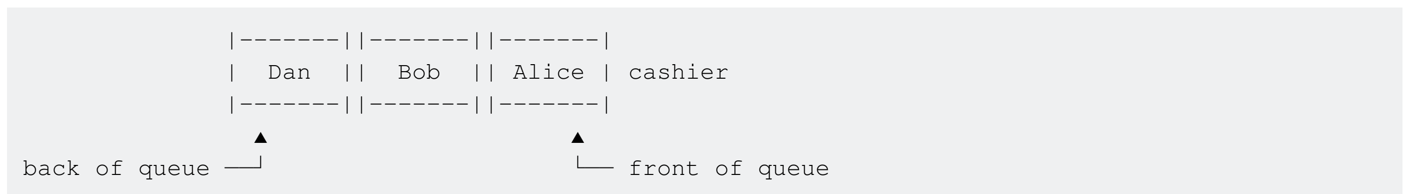
Consideremos el ejemplo de clientes que esperan ser atendidos. Alice, Bob y Dan están todos en el supermercado. Alice está lista para pagar, así que se acerca al cajero. Alice está ahora en la cola. Ella es la única persona en la cola, por lo que está tanto al frente como en la parte posterior.

Ahora, la cola se ve así:



```
               |-------|
               | Alice | cashier
               |-------|
                 ▲   ▲
back of queue ──┘   └── front of queue
```

Ahora Bob y luego Dan se acercan al cajero. Se agregan a la cola. Alice todavía está en la parte delantera, y Dan está en la parte posterior:



```
               |-------||-------||-------|
               |  Dan  ||  Bob  || Alice | cashier
               |-------||-------||-------|
                 ▲                     ▲
back of queue ──┘                     └── front of queue
```

Agregar una persona a la cola es la operación de puesta en cola. Alice, Bob y Dan han sido puestos en cola. A medida que el cajero ayude a cada cliente, se eliminarán de la cola. Esta es la operación de salida. Los clientes, que representan los elementos de datos en nuestra cola, se retiran de la cola de la cola. Esto significa que el primer cliente que se acercó al cajero fue el primer cliente que recibió ayuda (FIFO).

Implementación de la cola mediante el uso de matriz.

La cola sigue a FIFO como se menciona en la introducción. Cinco operaciones principales:

1. Encolar (x): empuja x a la parte posterior de la cola.
2. Dequeue (): muestra un elemento del frente de la cola.
3. isEmpty (): encuentra si la cola está vacía o no.
4. isFull (): encuentra si la cola está llena o no.
5. frontValue (): devuelve el valor frontal de la cola.

Todas las operaciones son constantes de tiempo O (1).

Código :

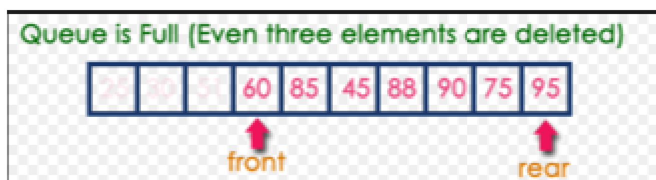
```
#include<stdio.h>

#define MAX 4

int front = -1;
int rear = -1;
int a[MAX];

bool isFull() {
  if(rear == MAX-1)
    return true;
  else
    return false;
}

bool isEmpty() {
  if(rear == -1 && front==-1)
    return true;
  else
    return false;
}

void enqueue(int data) {
  if (isFull()){
    printf("Queue is full\n");
    return;
  } else if(isEmpty()) {
    front = rear =0;
  } else {
    rear = rear + 1;
    a[rear] = data;
  }
}

void deque(){
  if(isEmpty()){
    printf("Queue is empty\n");
    return;
  } else if(front == rear) {
    front =-1;
    rear =-1;
  } else {
    front = front + 1;
  }
}

void print(){
  printf("Elements in Queue are\n");
  for(int i = front;i<=rear;i++){
    printf("%d ",a[i]);
  }
  printf("\n");
}

int frontValue(){
  printf("The front element after set of enqueue and dequeue is %d\n", a[front]);
}

int main(){
```

```
    deque();      // Queue is empty message will be thrown
    enqueue(10);
    print();
    enqueue(20);
    print();
    enqueue(30);
    print();
    enqueue(40);
    frontValue();
    print();
    enqueue(50);
    frontValue();
    deque();
    deque();
    enqueue(50);
    frontValue();
    print();
    return 0;
}
```

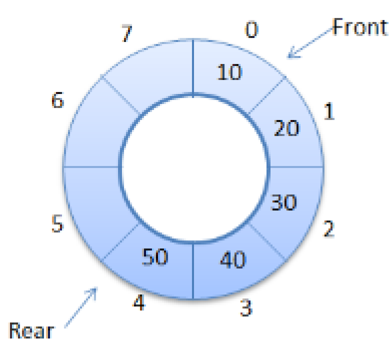
Implementación de una cola circular.

La memoria se organiza de manera eficiente en una cola circular en comparación con la cola lineal.

En cola lineal:



En cola circular:



Los espacios restantes pueden ser utilizados:

Código para que haga lo mismo:

```
#include<stdio.h>
#define MAX 10000
int front = -1;
int rear = -1;
int a[MAX];
```

```
bool isFull() {
  if((rear+1) % MAX == front)
    return true;
  else
    return false;
}

bool isEmpty() {
  if(rear == -1 && front==-1)
    return true;
  else
    return false;
}

void enqueue(int data) {
  if (isFull()){
    printf("Queue is full\n");
    return;
  } else if(isEmpty()) {
    front = rear = 0;
  } else {
    rear = (rear+1) % MAX;
    a[rear] = data;
  }
}

void deque() {
  if(isEmpty()){
    printf("Queue is empty\n");
    return;
  } else if(front == rear) {
    front =-1;
    rear =-1;
  } else {
    front = (front+1) % MAX;
  }
}

int frontValue() {
  return(a[front]);
}

int main() {
  deque();
  enqueue(10);
  enqueue(20);
  enqueue(30);
  enqueue(40);
  enqueue(50);
  frontValue();
  return 0;
}
```

Todas las operaciones tienen O (1) complejidad de tiempo.

Representación de lista enlazada de la cola

La representación de listas vinculadas es más eficiente en términos de gestión de memoria.

Código para mostrar encoue y deque en una cola usando la lista de enlaces en O (1).

```
#include<stdio.h>
#include<malloc.h>

struct node{
    int data;
    node* next;
};

node* front = NULL;
node* rear = NULL;

void enqueue(int data){      //adds element to end

    struct node* temp = (struct node*)malloc(sizeof(struct node*));
    temp->data = data;
    temp->next =NULL;

    if(front== NULL && rear == NULL){
        front = rear = temp;
        return;
    }
    rear->next = temp;
    rear= temp;
}

void deque(){     //removes element from front
    node* temp = front;

    if(front== NULL && rear == NULL){
        return;
    }
    else if (front==rear){
        front =rear = NULL;
    }
    else
    front= front ->next;

    free(temp);
}

void print(){
    node* temp = front;

    for(; temp != rear; temp=temp->next){
        printf("%d ",temp->data);
    }
    //for printing the rear element

        printf("%d ",temp->data);
    printf("\n");
}



int main(){
```

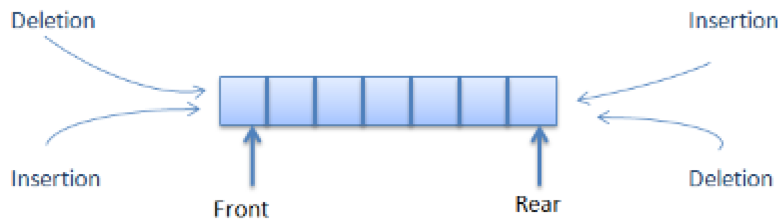
```
    enqueue(20);
    enqueue(50);
        enqueue(70);
        printf("After set of enques\n");
        print();

        deque();
        printf("After 1 deque\n");
        print();

        return 0;


}
```

Lea Cola en línea: https://riptutorial.com/es/data-structures/topic/7097/cola

Capítulo 6: Deque (doble cola de cola)

Examples

Inserción y eliminación tanto del frente como del final de la cola



Lea Deque (doble cola de cola) en línea: https://riptutorial.com/es/data-structures/topic/8849/deque--doble-cola-de-cola-

Capítulo 7: Estructura de datos de búsqueda de la unión

Introducción

Una estructura de datos de búsqueda de unión (o conjunto disjunto) es una estructura de datos simple, una partición de una serie de elementos en conjuntos disjuntos. Cada conjunto tiene un representante que se puede usar para distinguirlo de los otros conjuntos.

Se utiliza en muchos algoritmos, por ejemplo, para calcular árboles de expansión mínima a través del algoritmo de Kruskal, para calcular componentes conectados en gráficos no dirigidos y muchos más.

Examples

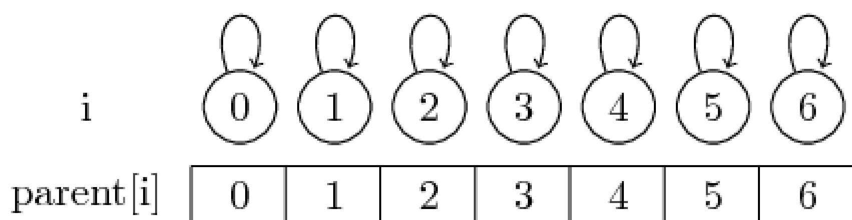
Teoría

Las estructuras de datos de búsqueda de unión proporcionan las siguientes operaciones:

- `make_sets(n)` inicializa una estructura de datos de búsqueda de unión con `n` singletons
- `find(i)` devuelve un representante para el conjunto del elemento `i`
- `union(i,j)` fusiona los conjuntos que contienen `i` y `j`

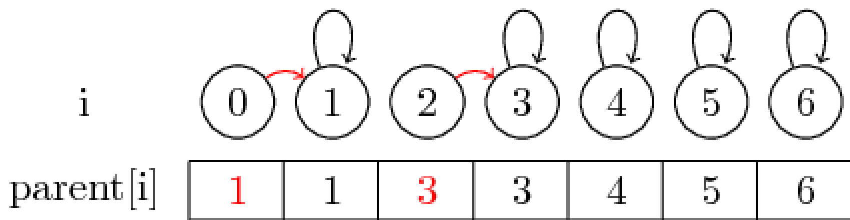
Nosotros representamos a nuestra partición de los elementos de `0` a `n - 1` mediante el almacenamiento de un elemento *padre* `parent[i]` para cada elemento `i` que finalmente conduce a un *representante* del conjunto que contiene `i` .
Si un elemento en sí es un representante, es su propio padre, es decir, el `parent[i] == i` .

Por lo tanto, si comenzamos con conjuntos singleton, cada elemento es su propio representante:

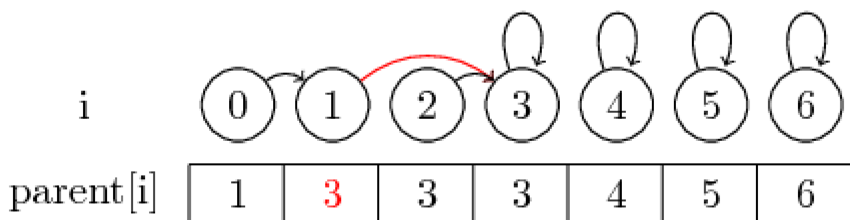


Podemos encontrar el representante para un conjunto dado simplemente siguiendo estos punteros principales.

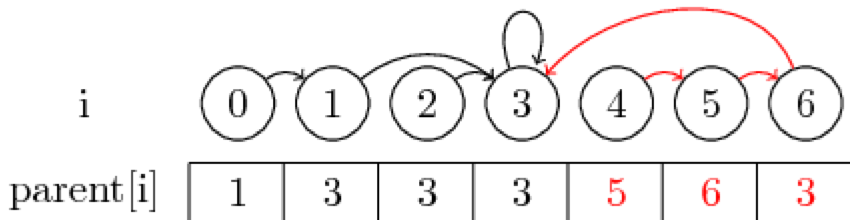
Veamos ahora cómo podemos unir conjuntos:
Si queremos fusionar los elementos 0 y 1 y los elementos 2 y 3, podemos hacer esto configurando el padre de 0 a 1 y configurando el padre de 2 a 3:



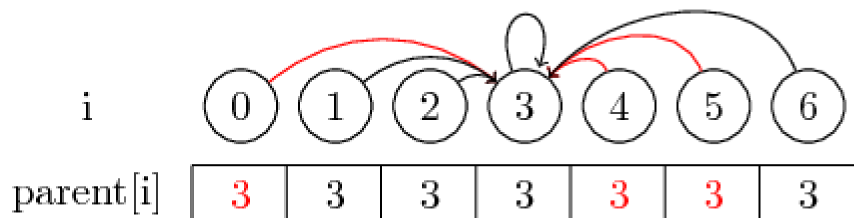
En este caso simple, solo se deben cambiar los elementos del puntero principal.
Sin embargo, si queremos combinar conjuntos más grandes, siempre debemos cambiar el puntero principal del *representante* del conjunto que se fusionará en otro conjunto: después de la **combinación (0,3)** , hemos establecido el principal del representante del conjunto Contiene 0 al representante del conjunto que contiene 3.



Para hacer el ejemplo un poco más interesante, ahora también **fusionamos (4,5), (5,6) y (3,4)** :



La última noción que quiero introducir es la **compresión de ruta** :
Si queremos encontrar el representante de un conjunto, es posible que tengamos que seguir varios indicadores *principales* antes de llegar al representante. Podríamos hacer esto más fácil almacenando el representante para cada conjunto directamente en su nodo principal. Perdemos el orden en el que fusionamos los elementos, pero potencialmente podemos tener una gran ganancia en tiempo de ejecución. En nuestro caso, las únicas rutas que no están comprimidas



son las rutas de 0, 4 y 5 a 3:

Implementacion basica

La implementación más básica de una estructura de datos de búsqueda de unión consiste en una matriz `parent` almacena un elemento principal para cada elemento de la estructura. Seguir estos 'punteros' primarios para un elemento `i` nos lleva al representante `j = find(i)` del conjunto que contiene `i` , donde retiene el `parent[j] = j` .

```
using std::size_t;

class union_find {
private:
    std::vector<size_t> parent;  // Parent for every node

public:
    union_find(size_t n) : parent(n) {
        for (size_t i = 0; i < n; ++i)
            parent[i] = i;      // Every element is its own representative
    }

    size_t find(size_t i) const {
        if (parent[i] == i)     // If we already have a representative
            return i;           // return it
        return find(parent[i]); // otherwise return the parent's representative
    }

    void merge(size_t i, size_t j) {
        size_t pi = find(i);
        size_t pj = find(j);
        if (pi != pj) {        // If the elements are not in the same set:
            parent[pi] = pj;   // Join the sets by marking pj as pi's parent
        }
    }
};
```

Mejoras: Compresión de ruta.

Si hacemos muchas operaciones de `merge` en una estructura de datos de búsqueda de unión, las rutas representadas por los punteros `parent` pueden ser bastante largas. *La compresión de ruta* , como ya se describió en la parte de teoría, es una forma simple de mitigar este problema.

Podríamos intentar hacer una compresión de ruta en toda la estructura de datos después de cada operación de fusión *k* -ésima o algo similar, pero tal operación podría tener un tiempo de ejecución bastante grande.

Por lo tanto, la compresión de ruta se usa principalmente en una pequeña parte de la estructura, especialmente en la ruta por la que caminamos para encontrar el representante de un conjunto. Esto se puede hacer almacenando el resultado de la operación de `find` después de cada subcall recursiva:

```
size_t find(size_t i) const {
    if (parent[i] == i)          // If we already have a representative
        return i;                // return it
    parent[i] = find(parent[i]); // path-compress on the way to the representative
    return parent[i];            // and return it
}
```

Mejoras: Unión por tamaño.

En nuestra implementación actual de `merge` , siempre elegimos que el conjunto izquierdo sea el hijo del conjunto correcto, sin tener en cuenta el tamaño de los conjuntos. Sin esta restricción, las rutas (sin *compresión de ruta*) de un elemento a su representante pueden ser bastante largas, lo que lleva a grandes tiempos de ejecución en las llamadas de `find` .

Otra mejora común es la *unión por tamaño* heurístico, que hace exactamente lo que dice: cuando se combinan dos conjuntos, siempre establecemos que el conjunto más grande es el padre del conjunto más pequeño, lo que lleva a una longitud de ruta de como máximo $\log n$ pasos:

Almacenamos un `std::vector<size_t> size` adicional de miembro `std::vector<size_t> size` en nuestra clase que se inicializa a 1 para cada elemento. Al fusionar dos conjuntos, el conjunto más grande se convierte en el padre del conjunto más pequeño y resumimos los dos tamaños:

```
private:
    ...
    std::vector<size_t> size;

public:
    union_find(size_t n) : parent(n), size(n, 1) { ... }

    ...

    void merge(size_t i, size_t j) {
        size_t pi = find(i);
        size_t pj = find(j);
        if (pi == pj) {             // If the elements are in the same set:
            return;                 // do nothing
        }
        if (size[pi] > size[pj]) { // Swap representatives such that pj
            std::swap(pi, pj);     // represents the larger set
        }
        parent[pi] = pj;           // attach the smaller set to the larger one
        size[pj] += size[pi];      // update the size of the larger set
    }
```

Mejoras: Unión por rango.

Otra heurística utilizada comúnmente en lugar de unión por tamaño es la *unión por rango* heurístico

Su idea básica es que en realidad no necesitamos almacenar el tamaño exacto de los conjuntos, una aproximación del tamaño (en este caso: aproximadamente el logaritmo del tamaño del conjunto) es suficiente para lograr la misma velocidad que la unión por tamaño.

Para esto, introducimos la noción del *rango* de un conjunto, que se da a continuación:

- Singletons tiene rango 0
- Si dos conjuntos con rango desigual se combinan, el conjunto con rango más alto se convierte en el padre mientras que el rango se mantiene sin cambios.
- Si se combinan dos conjuntos de igual rango, uno de ellos se convierte en el padre del otro

(la elección puede ser arbitraria), su rango se incrementa.

Una ventaja de la *unión por rango* es el uso del espacio: a medida que el rango máximo aumenta aproximadamente como $\log n$ Para todos los tamaños de entrada realistas, el rango se puede almacenar en un solo byte (ya que `n < 2^255`).

Una implementación simple de unión por rango podría verse así:

```
private:
    ...
    std::vector<unsigned char> rank;

public:
    union_find(size_t n) : parent(n), rank(n, 0) { ... }

    ...

    void merge(size_t i, size_t j) {
        size_t pi = find(i);
        size_t pj = find(j);
        if (pi == pj) {
            return;
        }
        if (rank[pi] < rank[pj]) {
            // link the smaller group to the larger one
            parent[pi] = pj;
        } else if (rank[pi] > rank[pj]) {
            // link the smaller group to the larger one
            parent[pj] = pi;
        } else {
            // equal rank: link arbitrarily and increase rank
            parent[pj] = pi;
            ++rank[pi];
        }
    }
```

Mejora final: Unión por rango con almacenamiento fuera de los límites

Mientras que en combinación con la compresión de ruta, union by rank casi logra operaciones de tiempo constante en estructuras de datos de union-find, hay un truco final que nos permite deshacernos del almacenamiento de `rank` almacenando el rango en las entradas fuera de los límites de la matriz `parent` . Se basa en las siguientes observaciones:

- En realidad, solo necesitamos almacenar el rango para los *representantes* , no para otros elementos. Para estos representantes, no necesitamos almacenar un `parent` .
- Hasta ahora, el `parent[i]` tiene un `size - 1` máximo de `size - 1` , es decir, no se utilizan los valores más grandes.
- Todos los rangos están a lo sumo $\log n$.

Esto nos lleva al siguiente enfoque:

- En lugar de la condición `parent[i] == i` , ahora identificamos representantes por `parent[i] >= size`
- Utilizamos estos valores fuera de límites para almacenar los rangos del conjunto, es decir, el

conjunto con el representante `i` tiene rango `parent[i] - size`
- Por lo tanto, inicializamos la matriz principal con `parent[i] = size` lugar de `parent[i] = i` , es decir, cada conjunto es su propio representante con rango 0.

Como solo compensamos los valores de rango por `size` , simplemente podemos reemplazar el vector de `rank` por el vector `parent` en la implementación de la `merge` y solo necesitamos intercambiar la condición que identifica a los representantes en `find` :

Implementación terminada usando unión por rango y compresión de ruta:

```
using std::size_t;

class union_find {
private:
    std::vector<size_t> parent;

public:
    union_find(size_t n) : parent(n, n) {} // initialize with parent[i] = n

    size_t find(size_t i) const {
        if (parent[i] >= parent.size()) // If we already have a representative
            return i;                   // return it
        return find(parent[i]);         // otherwise return the parent's repr.
    }

    void merge(size_t i, size_t j) {
        size_t pi = find(i);
        size_t pj = find(j);
        if (pi == pj) {
            return;
        }
        if (parent[pi] < parent[pj]) {
            // link the smaller group to the larger one
            parent[pi] = pj;
        } else if (parent[pi] > parent[pj]) {
            // link the smaller group to the larger one
            parent[pj] = pi;
        } else {
            // equal rank: link arbitrarily and increase rank
            parent[pj] = pi;
            ++parent[pi];
        }
    }
};
```

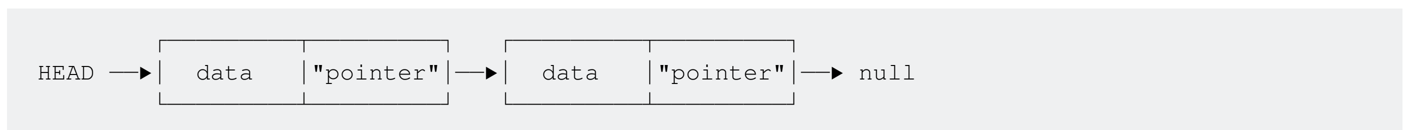
Lea Estructura de datos de búsqueda de la unión en línea: https://riptutorial.com/es/data-structures/topic/10684/estructura-de-datos-de-busqueda-de-la-union

Capítulo 8: Lista enlazada

Examples

Introducción a las listas enlazadas

Una lista enlazada es una colección lineal de elementos de datos, llamados nodos, que están vinculados a otro (s) nodo (s) mediante un "puntero". A continuación se muestra una lista enlazada individualmente con una referencia principal.



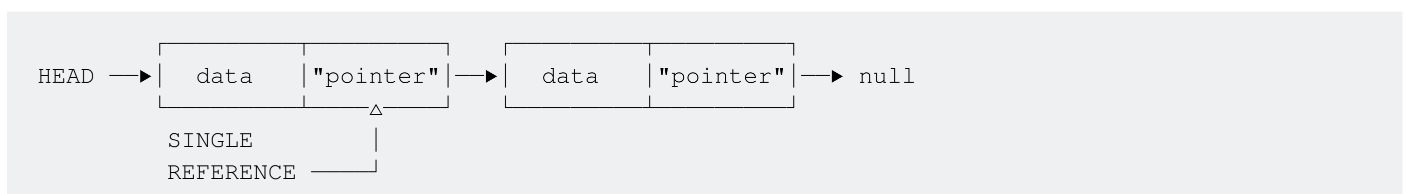
Hay muchos tipos de listas enlazadas, entre ellos por separado y doblemente listas enlazadas y listas enlazadas circulares.

Ventajas

- Las listas enlazadas son una estructura de datos dinámica, que puede aumentar y disminuir, asignando y desasignando memoria mientras el programa se está ejecutando.
- Las operaciones de inserción y eliminación de nodos se implementan fácilmente en una lista enlazada.
- Las estructuras de datos lineales, como pilas y colas, se implementan fácilmente con una lista vinculada.
- Las listas enlazadas pueden reducir el tiempo de acceso y pueden expandirse en tiempo real sin sobrecarga de memoria.

Lista enlazada individualmente

Las listas enlazadas individuales son un tipo de lista enlazada . Los nodos de una lista enlazada individualmente solo tienen un "puntero" a otro nodo, generalmente "siguiente". Se denomina lista enlazada individualmente porque cada nodo solo tiene un "puntero" a otro nodo. Una lista enlazada individualmente puede tener una referencia de cabecera y / o cola. La ventaja de tener una referencia de cola son los `getFromBack` , `addToBack` y `removeFromBack` , que se convierten en O (1).



Código de ejemplo en C

Código de ejemplo en Java, con pruebas unitarias - lista enlazada individualmente con referencia de cabecera

Lista enlazada XOR

Una **lista enlazada XOR** también se conoce como una **lista enlazada eficiente en memoria** . Es otra forma de una lista doblemente enlazada. Esto depende en gran medida de la puerta lógica **XOR** y sus propiedades.

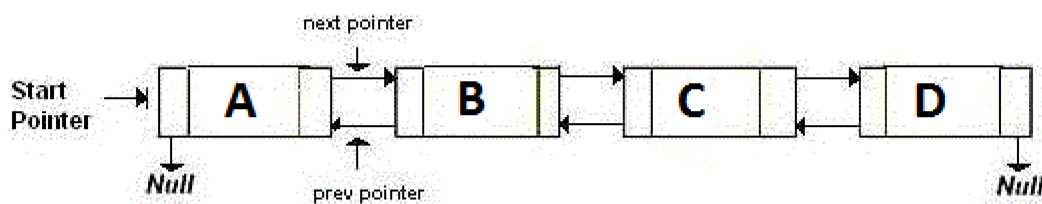
¿Por qué esto se llama la lista enlazada de memoria eficiente?

Esto se denomina así porque utiliza menos memoria que una lista tradicional doble vinculada.

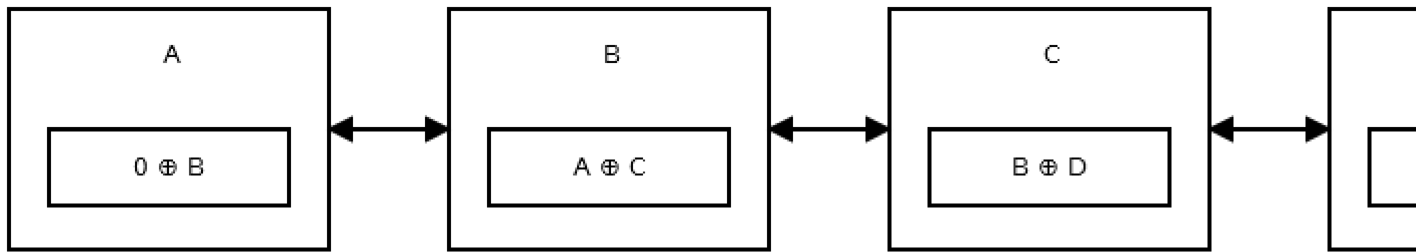
¿Es esto diferente de una lista doblemente vinculada?

Si lo es

Una **lista de** enlace doble almacena dos punteros, que apuntan al siguiente nodo y al anterior. Básicamente, si desea volver, vaya a la dirección señalada con el puntero `back` . Si desea avanzar, vaya a la dirección señalada por el `next` puntero. Es como:



Una **lista enlazada eficiente en memoria** , o la **lista enlazada XOR** solo tiene un puntero en lugar de dos. Esto almacena la dirección anterior (addr (anterior)) **XOR** (^) la siguiente dirección (addr (siguiente)). Cuando desee pasar al siguiente nodo, realice ciertos cálculos y busque la dirección del siguiente nodo. Esto es lo mismo para ir al nodo anterior. Es como:



¿Como funciona?

Para entender cómo funciona la lista enlazada, necesita conocer las propiedades de XOR (^):

```
|-------------|------------|------------|
|    Name     |   Formula  |   Result   |
|-------------|------------|------------|
| Commutative |    A ^ B   |   B ^ A    |
|-------------|------------|------------|
| Associative | A ^ (B ^ C)| (A ^ B) ^ C|
|-------------|------------|------------|
| None (1)    |    A ^ 0   |     A      |
|-------------|------------|------------|
| None (2)    |    A ^ A   |     0      |
|-------------|------------|------------|
| None (3)    | (A ^ B) ^ A|     B      |
|-------------|------------|------------|
```

Ahora dejemos esto de lado y veamos qué almacena cada nodo.

El primer nodo, o la **cabeza** , almacena `0 ^ addr (next)` ya que no hay ningún nodo o dirección anterior. Se parece a esto .

Luego, el segundo nodo almacena `addr (prev) ^ addr (next)` . Se parece a esto .

La **cola** de la lista, no tiene ningún nodo siguiente, por lo que almacena `addr (prev) ^ 0` . Se parece a esto .

Pasando de la cabeza al siguiente nodo

Digamos que ahora estás en el primer nodo, o en el nodo A. Ahora quieres moverte en el nodo B. Esta es la fórmula para hacerlo:

```
Address of Next Node = Address of Previous Node ^ pointer in the current Node
```

Así que esto sería:

```
addr (next) = addr (prev) ^ (0 ^ addr (next))
```

Como esta es la cabecera, la dirección anterior simplemente sería 0, así que:

```
addr (next) = 0 ^ (0 ^ addr (next))
```

Podemos eliminar los paréntesis:

```
addr (next) = 0 ^ 0 addr (next)
```

Usando la propiedad `none (2)` , podemos decir que `0 ^ 0` siempre será 0:

```
addr (next) = 0 ^ addr (next)
```

Usando la propiedad `none (1)` , podemos simplificarla para:

```
addr (next) = addr (next)
```

Tienes la dirección del siguiente nodo!

Pasando de un nodo al siguiente nodo

Ahora digamos que estamos en un nodo medio, que tiene un nodo anterior y el siguiente.

Apliquemos la fórmula:

```
Address of Next Node = Address of Previous Node ^ pointer in the current Node
```

Ahora sustituye los valores:

```
addr (next) = addr (prev) ^ (addr (prev) ^ addr (next))
```

Eliminar paréntesis:

```
addr (next) = addr (prev) ^ addr (prev) ^ addr (next)
```

Usando la propiedad `none (2)` , podemos simplificar:

```
addr (next) = 0 ^ addr (next)
```

Usando la propiedad `none (1)` , podemos simplificar:

```
addr (next) = addr (next)
```

Y lo entiendes!

Pasando de un nodo al nodo en el que estaba anteriormente

Si no entiende el título, básicamente significa que si estuvo en el nodo X y ahora se ha movido al

nodo Y, desea volver al nodo visitado anteriormente, o básicamente al nodo X.

Esto no es una tarea engorrosa. Recuerde que mencioné anteriormente, que almacena la dirección en la que se encontraba en una variable temporal. Por lo tanto, la dirección del nodo que desea visitar está en una variable:

```
addr (prev) = temp_addr
```

Mover de un nodo al nodo anterior

Esto no es lo mismo que se mencionó anteriormente. Quiero decir que estabas en el nodo Z, ahora estás en el nodo Y y quieres ir al nodo X.

Esto es casi lo mismo que pasar de un nodo al siguiente nodo. Solo que esto es viceversa. Cuando escriba un programa, utilizará los mismos pasos que mencioné al pasar de un nodo al siguiente, solo que está encontrando el elemento anterior en la lista que el elemento siguiente.

Código de ejemplo en C

```
/* C/C++ Implementation of Memory efficient Doubly Linked List */
#include <stdio.h>
#include <stdlib.h>

// Node structure of a memory efficient doubly linked list
struct node
{
    int data;
    struct node* npx;  /* XOR of next and previous node */
};

/* returns XORed value of the node addresses */
struct node* XOR (struct node *a, struct node *b)
{
    return (struct node*) ((unsigned int) (a) ^ (unsigned int) (b));
}

/* Insert a node at the begining of the XORed linked list and makes the
   newly inserted node as head */
void insert(struct node **head_ref, int data)
{
    // Allocate memory for new node
    struct node *new_node  = (struct node *) malloc (sizeof (struct node) );
    new_node->data = data;

    /* Since new node is being inserted at the begining, npx of new node
       will always be XOR of current head and NULL */
    new_node->npx = XOR(*head_ref, NULL);

    /* If linked list is not empty, then npx of current head node will be XOR
       of new node and node next to current head */
    if (*head_ref != NULL)
    {
        // *(head_ref)->npx is XOR of NULL and next. So if we do XOR of
```

```
        // it with NULL, we get next
        struct node* next = XOR((*head_ref)->npx,  NULL);
        (*head_ref)->npx = XOR(new_node, next);
    }

    // Change head
    *head_ref = new_node;
}

// prints contents of doubly linked list in forward direction
void printList (struct node *head)
{
    struct node *curr = head;
    struct node *prev = NULL;
    struct node *next;

    printf ("Following are the nodes of Linked List: \n");

    while (curr != NULL)
    {
        // print current node
        printf ("%d ", curr->data);

        // get address of next node: curr->npx is next^prev, so curr->npx^prev
        // will be next^prev^prev which is next
        next = XOR (prev, curr->npx);

        // update prev and curr for next iteration
        prev = curr;
        curr = next;
    }
}

// Driver program to test above functions
int main ()
{
    /* Create following Doubly Linked List
       head-->40<-->30<-->20<-->10   */
    struct node *head = NULL;
    insert(&head, 10);
    insert(&head, 20);
    insert(&head, 30);
    insert(&head, 40);

    // print the created list
    printList (head);

    return (0);
}
```

El código anterior realiza dos funciones básicas: inserción y transversal.

- **Inserción:** Esto se realiza mediante la función `insert` . Esto inserta un nuevo nodo al principio. Cuando se llama a esta función, coloca el nuevo nodo como cabecera y el nodo anterior como segundo nodo.
- **Traversal:** Esto se realiza mediante la función `printList` . Simplemente pasa por cada nodo e imprime el valor.

Tenga en cuenta que XOR de punteros no está definido por el estándar C / C ++. Por lo tanto, la implementación anterior puede no funcionar en todas las plataformas.

Referencias

- https://cybercruddotnet.wordpress.com/2012/07/04/complicating-things-with-xor-linked-lists/
- http://www.ritambhara.in/memory-efficient-doubly-linked-list/comment-page-1/
- http://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-2/

Tenga en cuenta que he tomado un montón de contenido de mi propia respuesta en main.

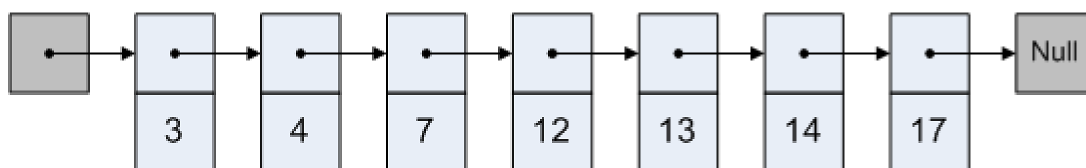
Omitir lista

Las listas de omisiones son listas vinculadas que le permiten saltar al nodo correcto. Este es un método que es mucho más rápido que una lista enlazada individualmente normal. Básicamente es una lista enlazada individualmente, pero los punteros no van de un nodo al siguiente, sino que omiten algunos nodos. De ahí el nombre "Skip List".

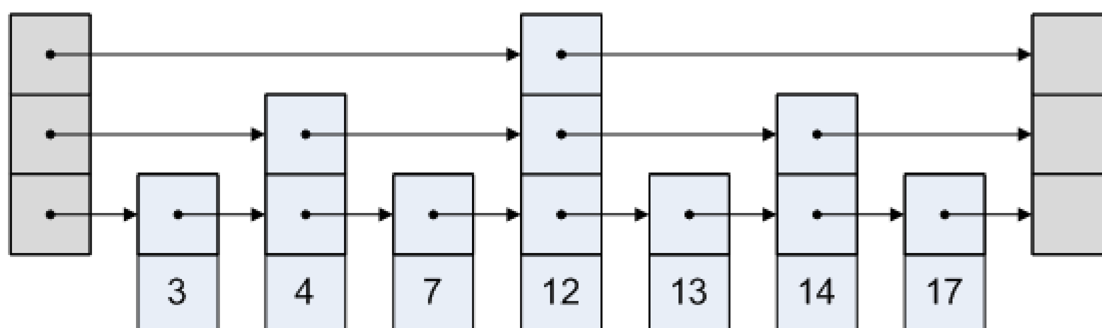
¿Es esto diferente de una lista enlazada individualmente?

Si lo es

Una lista enlazada individualmente es una lista en la que cada nodo apunta al siguiente nodo. Una representación gráfica de una lista enlazada individualmente es como:



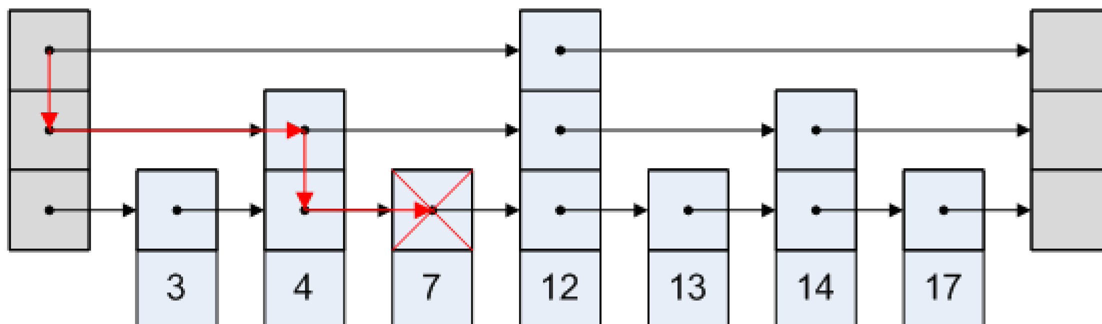
Una lista de omisiones es una lista con cada nodo que apunta a un nodo que puede o no estar detrás de él. Una representación gráfica de una lista de omisión es:



¿Como funciona?

La lista de saltos es simple. Digamos que queremos acceder al nodo 3 en la imagen de arriba. No podemos tomar la ruta de ir de la cabeza al cuarto nodo, ya que está despiués del tercer nodo. Entonces vamos de la cabeza al segundo nodo, y luego al tercero.

Una representación gráfica es como:

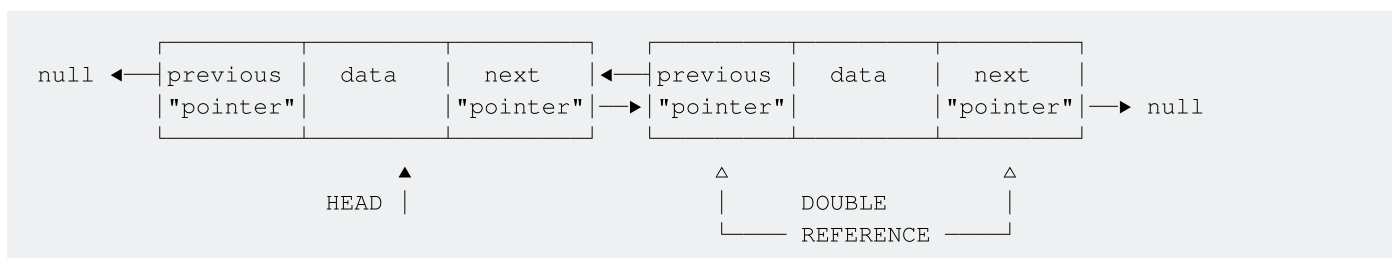


Referencias

- http://igoro.com/archive/skip-lists-are-fascinating/

Lista doblemente vinculada

Las listas enlazadas doblemente son un tipo de lista enlazada . Los nodos de una lista doblemente enlazada tienen dos "punteros" a otros nodos, "siguiente" y "anterior". Se llama lista enlazada doble porque cada nodo solo tiene dos "punteros" a otros nodos. Una lista doblemente enlazada puede tener un puntero de cabeza y / o cola.



Las listas con enlaces dobles son menos eficientes en espacio que las listas con enlaces individuales; sin embargo, para algunas operaciones, ofrecen mejoras significativas en la eficiencia de tiempo. Un ejemplo simple es la función `get` , que para una lista doblemente enlazada con una referencia de cabecera y cola comenzará desde la cabecera o la cola, dependiendo del índice del elemento que se obtenga. Para una lista de elementos `n` , para obtener el elemento indexado `n/2 + i` , una lista enlazada individualmente con referencias de cabecera / cola debe atravesar los nodos `n/2 + i` , porque no puede "retroceder" desde la cola. Una lista doblemente enlazada con referencias cabeza / cola solo tiene que atravesar `n/2 - i` nodos, ya que puede "retroceder" desde la cola, atravesando la lista en orden inverso.

Código de ejemplo en C

Un ejemplo básico de SinglyLinkedList en Java

Una implementación básica para la lista enlazada individualmente en java - que puede agregar valores enteros al final de la lista, eliminar el primer valor encontrado del valor, devolver una matriz de valores en un instante dado y determinar si un valor dado está presente en la lista.

Node.java

```
package com.example.so.ds;


/**
 * <p> Basic node implementation </p>
 * @since 20161008
 * @author Ravindra HV
 * @version 0.1
 */
public class Node {

    private Node next;
    private int value;

    public Node(int value) {
        this.value=value;
    }

    public Node getNext() {
        return next;
    }

    public void setNext(Node next) {
        this.next = next;
    }

    public int getValue() {
        return value;
    }


}
```

SinglyLinkedList.java

```
package com.example.so.ds;


/**
 * <p> Basic single-linked-list implementation </p>
 * @since 20161008
 * @author Ravindra HV
 * @version 0.2
 */
public class SinglyLinkedList {

    private Node head;
```

```
    private volatile int size;

    public int getSize() {
        return size;
    }

    public synchronized void append(int value) {

        Node entry = new Node(value);
        if(head == null) {
            head=entry;
        }
        else {
            Node temp=head;
            while( temp.getNext() != null) {
                temp=temp.getNext();
            }
            temp.setNext(entry);
        }

        size++;
    }


    public synchronized boolean removeFirst(int value) {
        boolean result = false;

        if( head == null ) { // or size is zero..
            // no action
        }
        else if( head.getValue() == value ) {
            head = head.getNext();
            result = true;
        }
        else {

            Node previous = head;
            Node temp = head.getNext();
            while( (temp != null) && (temp.getValue() != value) ) {
                previous = temp;
                temp = temp.getNext();
            }

            if((temp != null) && (temp.getValue() == value)) { // if temp is null then not
found..
                previous.setNext( temp.getNext() );
                result = true;
            }

        }

        if(result) {
            size--;
        }

        return result;

    }

    public synchronized int[] snapshot() {
        Node temp=head;
```

```
        int[] result = new int[size];
        for(int i=0;i<size;i++) {
            result[i]=temp.getValue();
            temp = temp.getNext();
        }
        return result;
    }

    public synchronized boolean contains(int value) {
        boolean result = false;
        Node temp = head;

        while(temp!=null) {
            if(temp.getValue() == value) {
                result=true;
                break;
            }
            temp=temp.getNext();
        }
        return result;
    }

}
```

TestSinglyLinkedList.java

```
package com.example.so.ds;

import java.util.Arrays;
import java.util.Random;


/**
 *
 * <p> Test-case for single-linked list</p>
 * @since 20161008
 * @author Ravindra HV
 * @version 0.2
 *
 */
public class TestSinglyLinkedList {

    /**
     * @param args
     */
    public static void main(String[] args) {

        SinglyLinkedList singleLinkedList = new SinglyLinkedList();

        int loop = 11;
        Random random = new Random();

        for(int i=0;i<loop;i++) {

            int value = random.nextInt(loop);
            singleLinkedList.append(value);

            System.out.println();
            System.out.println("Append :"+value);
            System.out.println(Arrays.toString(singleLinkedList.snapshot()));
```

```
            System.out.println(singleLinkedList.getSize());
            System.out.println();
        }

        for(int i=0;i<loop;i++) {
            int value = random.nextInt(loop);
            boolean contains = singleLinkedList.contains(value);
            singleLinkedList.removeFirst(value);

            System.out.println();
            System.out.println("Contains :"+contains);
            System.out.println("RemoveFirst :"+value);
            System.out.println(Arrays.toString(singleLinkedList.snapshot()));
            System.out.println(singleLinkedList.getSize());
            System.out.println();
        }


    }

}
```

Lea Lista enlazada en línea: https://riptutorial.com/es/data-structures/topic/2967/lista-enlazada

Capítulo 9: Matrices

Observaciones

Este artículo pretende explicar qué es una matriz y cómo usarla.

Examples

Introducción a las matrices.

Las matrices son esencialmente matrices bidimensionales.

Eso significa que asocia las coordenadas (i, j), donde i es la fila y j es la columna, a un valor.

Entonces, podrías tener:

$m_{3, 4}$ = "Hola"

La implementación más fácil es hacer una matriz o matrices. En python, que iría de la siguiente manera.

```
matrix = [["a", "b", "c"], ["d", "e", "f"], ["g", "h", "i"]]
print(matrix[1][2]) #returns "f"
```

Lea Matrices en línea: https://riptutorial.com/es/data-structures/topic/7455/matrices

Capítulo 10: Montón binario

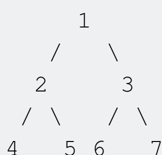
Introducción

Un montón binario es un árbol binario completo que satisface la propiedad de ordenamiento del montón. El orden puede ser uno de dos tipos: la propiedad min-heap: el valor de cada nodo es mayor o igual que el valor de su padre, con el elemento de valor mínimo en la raíz.

Examples

Ejemplo

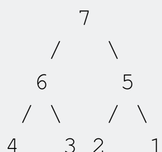
Min-montón



```
       1
     /   \
    2     3
   / \   / \
  4   5 6   7
```

El árbol anterior es un Min-Heap ya que la raíz es el mínimo entre todos los nodos presentes en el árbol. La misma propiedad es seguida por todos los nodos en el árbol.

Max-Heap



```
       7
     /   \
    6     5
   / \   / \
  4   3 2   1
```

El árbol anterior es un Max-Heap ya que la raíz es el máximo entre todos los nodos presentes en el árbol. La misma propiedad es seguida por todos los nodos en el árbol.

Operaciones soportadas por Min-Heap

1. **getMin ()** : devuelve el elemento raíz. Ya que es el primer elemento de una matriz, podemos recuperar el elemento mínimo en `O(1)`

2. **extractMin ()** : elimina el elemento mínimo del montón. Una vez que elimina el elemento, el árbol debe satisfacer la propiedad Min-Heap para que se **realice** una operación (**heapifying**) para mantener la propiedad del árbol `O(logn)` toma `O(logn)`

3. **dereaseKey ()** - Disminuye el valor de la clave. La complejidad del tiempo para esta operación es `O(logn)`

4. **Insertar ()** : la clave siempre se inserta al final del árbol. Si la clave agregada no sigue la propiedad del montón, entonces debemos filtrarla para que el árbol satisfaga la propiedad

del montón. Este paso toma `O(logn)` hora.

5. **delete ()** : este paso lleva tiempo `O(logn)` Para eliminar una clave, primero debemos reducir el valor de la clave a un valor mínimo y luego extraer este valor mínimo.

Aplicación de Heap

1. Heap Sort
2. Cola de prioridad

Heap utiliza muchos de los algoritmos de gráficos como `Dijkstra's Shortest Path` y `Prim's Minimum Spanning Tree` .

Implementación en Java

```
public class MinHeap {

    int hArr[];
    int capacity;
    int heapSize;

    public MinHeap(int capacity){
        this.heapSize = 0;
        this.capacity = capacity;
        hArr = new int[capacity];
    }

    public int getparent(int i){
        return (i-1)/2;
    }

    public int getLeftChild(int i){
        return 2*i+1;
    }

    public int getRightChild(int i){
        return 2*i+2;
    }

    public void insertKey(int k){
        if(heapSize==capacity)
            return;

        heapSize++;
        int i = heapSize-1;
        hArr[i] = k;

        while(i!=0 && hArr[getparent(i)]>hArr[i]){
            swap(hArr[i],hArr[getparent(i)]);
            i = getparent(i);
        }
    }

    public int extractMin(){
        if(heapSize==0)
            return Integer.MAX_VALUE;

        if(heapSize==1){
```

```
            heapSize--;
            return hArr[0];
        }

        int root = hArr[0];
        hArr[0] = hArr[heapSize-1];
        heapSize--;
        MinHeapify(0);

        return root;
    }

    public void decreaseKey(int i , int newVal){
        hArr[i] = newVal;
        while(i!=0 && hArr[getparent(i)]>hArr[i]){
            swap(hArr[i],hArr[getparent(i)]);
            i = getparent(i);
        }
    }

    public void deleteKey(int i){
        decreaseKey(i, Integer.MIN_VALUE);
        extractMin();
    }

    public void MinHeapify(int i){
        int l = getLeftChild(i);
        int r = getRightChild(i);
        int smallest = i;
        if(l<heapSize && hArr[l] < hArr[i])
            smallest = l;
        if(l<heapSize && hArr[r] < hArr[smallest])
            smallest = r;

        if(smallest!=i){
            swap(hArr[i], hArr[smallest]);
            MinHeapify(smallest);
        }
    }

    public void swap(int x, int y){
        int temp = hArr[x];
        hArr[x] = hArr[y];
        hArr[y] = temp;
    }
}
```

Lea Montón binario en línea: https://riptutorial.com/es/data-structures/topic/10799/monton-binario

Capítulo 11: Recorridos de grafos

Introducción

Todos los algoritmos relacionados con los recorridos gráficos. Sus complejidades, tanto en tiempo de ejecución como en espacio.

Examples

Primera búsqueda de profundidad

La profundidad del primer recorrido (o búsqueda) para un gráfico es similar a la profundidad del primer recorrido de un árbol. La única captura aquí es que, a diferencia de los árboles, los gráficos pueden contener ciclos, por lo que podemos llegar al mismo nodo nuevamente. Para evitar procesar un nodo más de una vez, usamos una matriz visitada booleana.

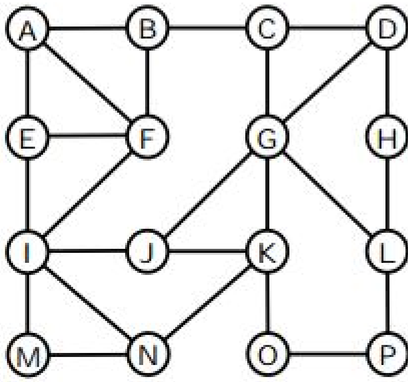
El siguiente algoritmo presenta los pasos para el recorrido del gráfico utilizando DFS:

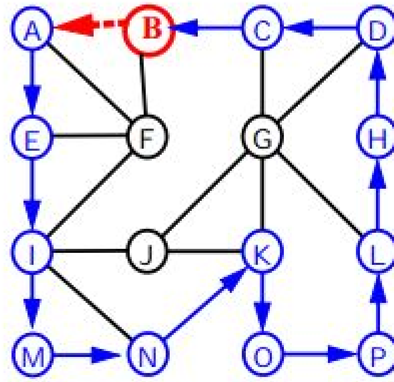
Algoritmo DFS (v);

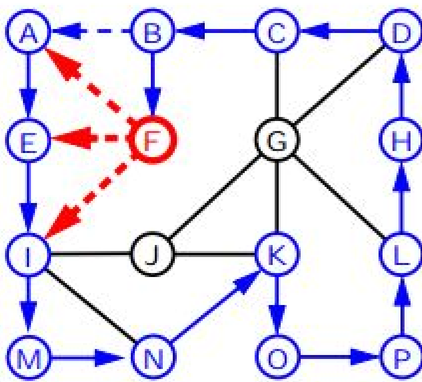
Entrada : un vértice v en una gráfica

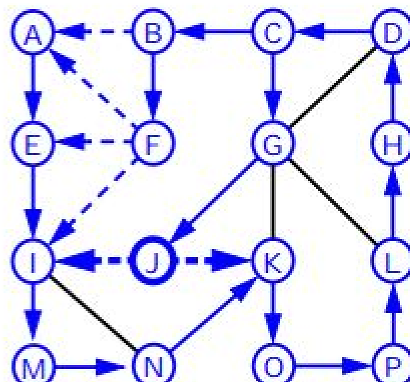
Salida : Un etiquetado de los bordes como bordes de "descubrimiento" y "aristas".

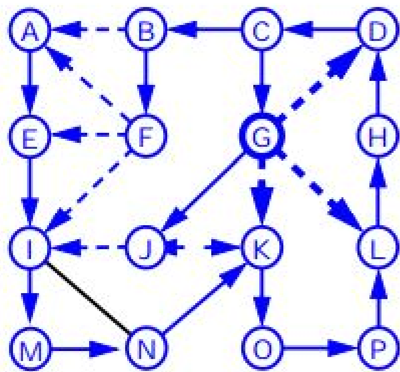
```
for each edge e incident on v do

    if edge e is unexplored then

    let w be the other endpoint of e
    if vertex w is unexplored then
        label e as a discovery edge
        recursively call DFS(w)
    else
    label e as a backedge
```

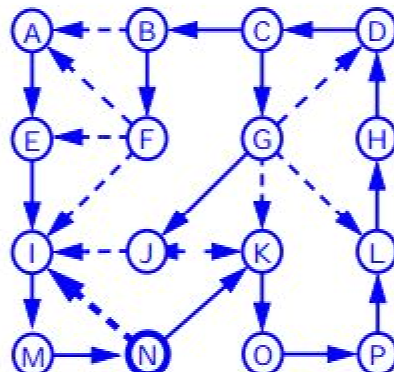












Amplia primera búsqueda

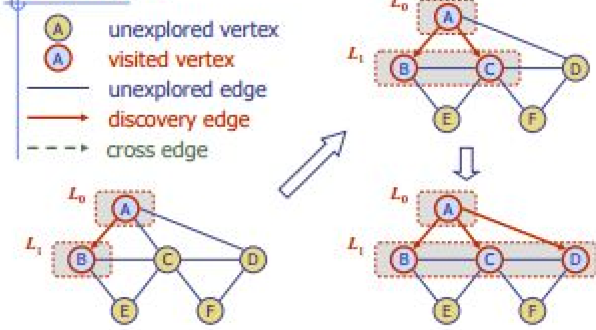
Algoritmo BFS (G)

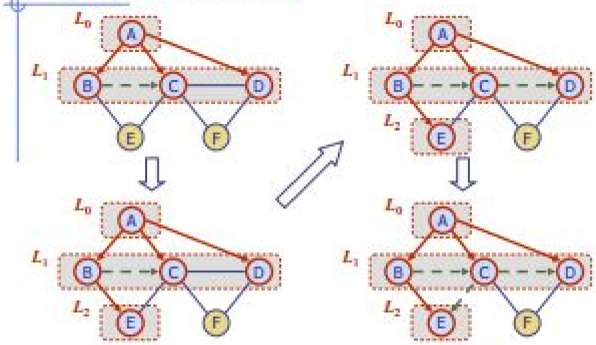
Gráfico de **entrada G**

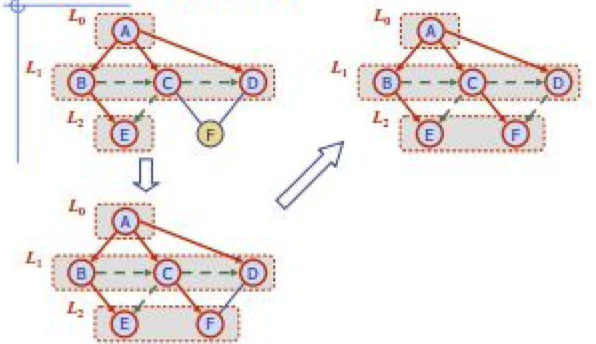
Etiquetado de **salida** de los bordes y partición de los vértices de **G.**

```
for all u ∈ G.vertices()
    setLabel(u, UNEXPLORED)
```

```
    for all e ∈ G.edges()
    setLabel
    (e, UNEXPLORED)
    for all v ∈ G.vertices()
        if getLabel(v) = UNEXPLORED
            BFS(G, v)
```







Lea Recorridos de grafos en línea: https://riptutorial.com/es/data-structures/topic/8688/recorridos-de-grafos

Capítulo 12: Trie (árbol de prefijo / árbol de radix)

Examples

Introducción a Trie

¿Te has preguntado alguna vez cómo funcionan los motores de búsqueda? ¿Cómo alinea Google millones de resultados frente a usted en tan solo unos pocos milisegundos? ¿Cómo una enorme base de datos situada a miles de kilómetros de usted encuentra la información que está buscando y se la envía? La razón detrás de esto no es posible solo mediante el uso más rápido de Internet y superordenadores. Algunos algoritmos de búsqueda fascinantes y estructuras de datos funcionan detrás de esto. Una de ellas es Trie .

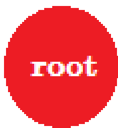
Trie , también llamado *árbol digital* y, a veces, *árbol de prefijos* o de *radix* (ya que pueden buscarse por prefijos), es un tipo de árbol de búsqueda: una estructura de datos de árbol ordenada que se utiliza para almacenar un conjunto dinámico o una matriz asociativa donde están las claves. por lo general cuerdas. Es una de esas estructuras de datos que se pueden implementar fácilmente. Digamos que tienes una enorme base de datos de millones de palabras. Puede usar trie para almacenar esta información y la complejidad de la búsqueda de estas palabras depende solo de la longitud de la palabra que estamos buscando. Eso significa que no depende de qué tan grande es nuestra base de datos. ¿No es eso asombroso?

Supongamos que tenemos un diccionario con estas palabras:

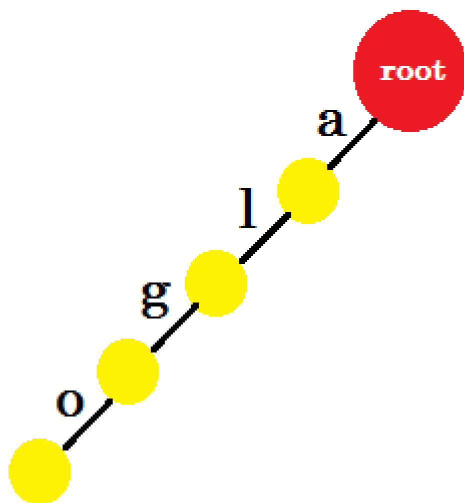
```
algo
algea
also
tom
to
```

Queremos almacenar este diccionario en la memoria de tal manera que podamos encontrar fácilmente la palabra que estamos buscando. Uno de los métodos consistiría en clasificar las palabras lexicográficamente: cómo los diccionarios de la vida real almacenan palabras. Entonces podemos encontrar la palabra haciendo una *búsqueda binaria* . Otra forma es usar **Prefix Tree** o **Trie** , en resumen. La palabra **'Trie'** viene de la palabra Re **trie** val. Aquí, **prefijo** denota *el prefijo de cadena* que puede definirse así: Todas las subcadenas que pueden crearse desde el principio de una cadena se llaman prefijo. Por ejemplo: 'c', 'ca', 'cat' son todos el prefijo de la cadena 'cat'.

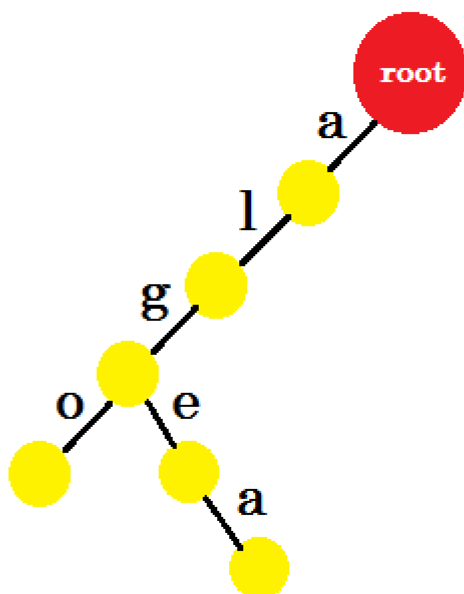
Ahora volvamos a **Trie** . Como su nombre lo sugiere, crearemos un árbol. Al principio, tenemos un árbol vacío con solo la raíz:



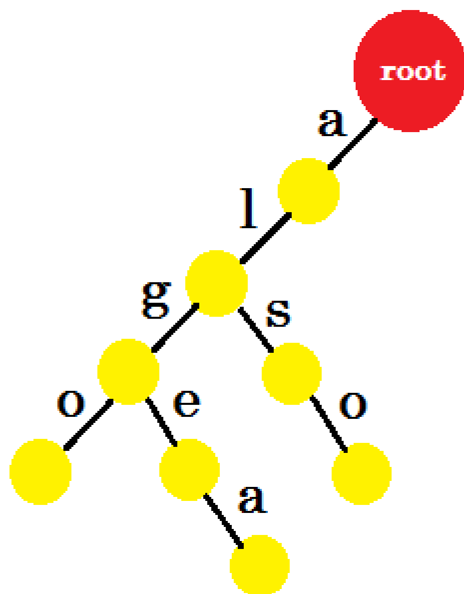
Insertaremos la palabra **'algo'** . Crearemos un nuevo nodo y nombraremos el borde entre estos dos nodos **'a'** . Desde el nuevo nodo crearemos otra ventaja llamada **'l'** y la conectaremos con otro nodo. De esta manera, crearemos dos nuevos bordes para **'g'** y **'o'** . Tenga en cuenta que no estamos almacenando ninguna información en los nodos. Por ahora, solo consideraremos crear nuevos bordes desde estos nodos. A partir de ahora, vamos a llamar a un borde llamado **'x'** - **edge-x**



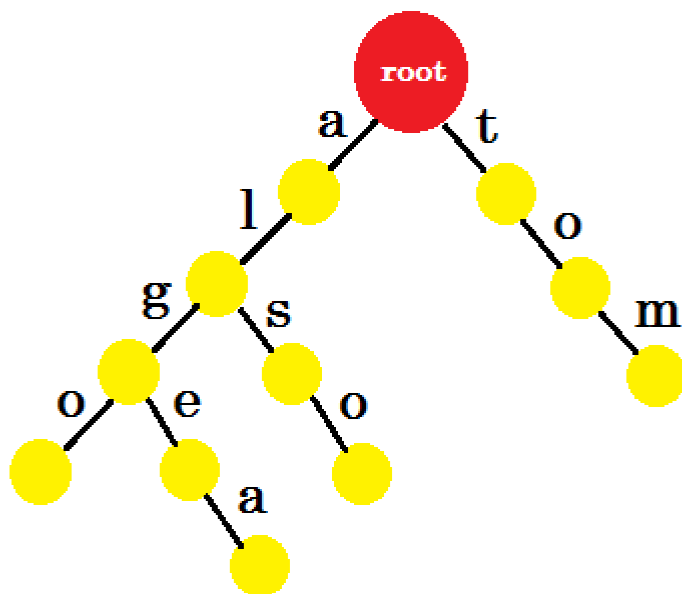
Ahora queremos añadir la palabra **'algea'** . Necesitamos un **edge-a** desde la raíz, que ya tenemos. Así que no necesitamos agregar nuevas ventajas. De manera similar, tenemos un borde de **'a'** a **'l'** y **'l'** a **'g'** . Eso significa que **'alg'** ya está en el **trie** . Solo agregaremos **edge-e** y **edge-a** con él.



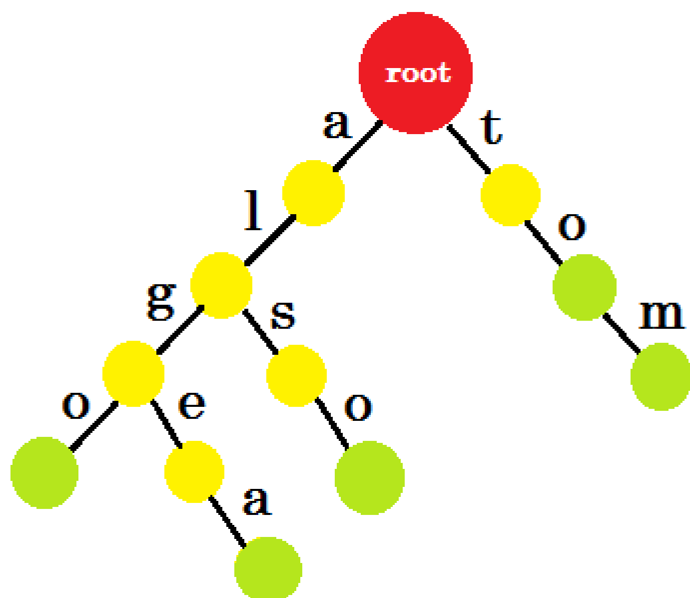
Añadiremos la palabra **'también'** . Tenemos el prefijo **'al'** de root. Sólo añadiremos **'así'** con él.



Vamos a añadir **'tom'** . Esta vez, creamos un nuevo borde desde la raíz ya que no tenemos ningún prefijo de tom creado anteriormente.



Ahora, ¿cómo debemos agregar **'a'** ? **'to'** es completamente un prefijo de **'tom'** , por lo que no necesitamos agregar ninguna ventaja. Lo que podemos hacer es poner marcas finales en algunos nodos. Pondremos marcas finales en aquellos nodos donde se complete al menos una palabra. El círculo verde denota la marca final. El trie se verá como:



Puedes entender fácilmente por qué agregamos las marcas finales. Podemos determinar las palabras almacenadas en trie. Los caracteres estarán en los bordes y los nodos contendrán las marcas finales.

Ahora puedes preguntar, ¿cuál es el propósito de almacenar palabras como esta? Digamos, se le pide que encuentre la palabra **'alice'** en el diccionario. Vas a atravesar el trie. Comprobarás si hay un **edge-a** desde la raíz. A continuación, compruebe desde **'a'** , si hay un **borde-l** . Después de eso, no encontrará ninguna **edge-i** , por lo que puede llegar a la conclusión de que la palabra **alice** no existe en el diccionario.

Si se le pide que busque la palabra **'alg'** en el diccionario, atravesará **root-> a** , **a-> l** , **l-> g** , pero no encontrará un nodo verde al final. Así que la palabra no existe en el diccionario. Si busca **'tom'** , terminará en un nodo verde, lo que significa que la palabra existe en el diccionario.

Complejidad:

La cantidad máxima de pasos necesarios para buscar una palabra en un **trío** es la longitud de la palabra que estamos buscando. La complejidad es **O (longitud)** . La complejidad para la inserción es la misma. La cantidad de memoria necesaria para implementar **trie** depende de la implementación. Veremos una implementación en otro ejemplo donde podemos almacenar $\mathbf{10^{6}}$ caracteres (no palabras, letras) en un **trie** .

Uso de Trie:

- Para insertar, eliminar y buscar una palabra en el diccionario.
- Para averiguar si una cadena es un prefijo de otra cadena.
- Para saber cuántas cadenas tiene un prefijo común.
- Sugerencia de nombres de contacto en nuestros teléfonos dependiendo del prefijo que ingresamos.
- Averiguar 'Substring Común más largo' de dos cadenas.
- Averiguar la longitud del 'Prefijo común' para dos palabras usando 'El ancestro común más largo'

Implementación de Trie

Antes de leer este ejemplo, es muy recomendable que lea Introducción a Trie primero.

Una de las maneras más fáciles de implementar **Trie** es usar la lista enlazada.

Nodo:

Los nodos constarán de:

1. Variable para la marca final.
2. Pointer Array al siguiente nodo.

La variable marca de final simplemente indicará si es un punto final o no. La matriz de punteros indicará todos los bordes posibles que se pueden crear. Para los alfabetos en inglés, el tamaño de la matriz será de 26, ya que se pueden crear un máximo de 26 bordes desde un solo nodo. Al principio, cada valor de la matriz de punteros será NULL y la variable de marca final será falsa.

```
Node:
Boolean Endmark
Node *next[26]
Node()
    endmark = false
    for i from 0 to 25
        next[i] := NULL
    endfor
endNode
```

Cada elemento en la **siguiente []** matriz apunta a otro nodo. **el siguiente [0]** apunta al nodo que comparte el **borde-a** , el **siguiente [1]** apunta al nodo que comparte el **borde-b** y así sucesivamente. Hemos definido el constructor de Nodo para inicializar Endmark como falso y poner NULL en todos los valores de la **siguiente** matriz de punteros **[]** .

Para crear **Trie** , primero necesitaremos crear una instancia de **root** . Luego, desde la **raíz** , crearemos otros nodos para almacenar información.

Inserción:

```
Procedure Insert(S, root):    // s is the string that needs to be inserted in our dictionary
curr := root
for i from 1 to S.length
    id := S[i] - 'a'
    if curr -> next[id] == NULL
        curr -> next[id] = Node()
    curr := curr -> next[id]
endfor
curr -> endmark = true
```

Aquí estamos trabajando con **az** . Entonces, para convertir sus valores ASCII a **0-25** , restamos el valor ASCII de **'a'** de ellos. Comprobaremos si el nodo actual (curr) tiene un borde para el personaje en cuestión. Si lo hace, nos movemos al siguiente nodo usando ese borde, si no creamos un nuevo nodo. Al final, cambiamos la marca final a verdadero.

Buscando:

```
Procedure Search(S, root)      // S is the string we are searching
curr := root
for i from 1 to S.length
    id := S[i] - 'a'
    if curr -> next[id] == NULL
        Return false
    curr := curr -> id
Return curr -> endmark
```

El proceso es similar a insertar. Al final, devolvemos la **marca final** . De modo que si la **marca final** es verdadera, eso significa que la palabra existe en el diccionario. Si es falso, entonces la palabra no existe en el diccionario.

Esta fue nuestra principal implementación. Ahora podemos insertar cualquier palabra en **trie** y buscarla.

Supresión:

A veces necesitaremos borrar las palabras que ya no se utilizarán para ahorrar memoria. Para este propósito, necesitamos eliminar las palabras no utilizadas:

```
Procedure Delete(curr):       //curr represents the current node to be deleted
for i from 0 to 25
    if curr -> next[i] is not equal to NULL
        delete(curr -> next[i])
    delete(curr)               //free the memory the pointer is pointing to
```

Esta función va a todos los nodos secundarios de **curr** , los elimina y luego se elimina **curr** para liberar la memoria. Si llamamos **delete (root)** , eliminará todo **Trie** .

Trie puede implementarse usando *Arrays* también.

Lea Trie (árbol de prefijo / árbol de radix) en línea: https://riptutorial.com/es/data-structures/topic/7178/trie--arbol-de-prefijo---arbol-de-radix-

Creditos

S. No	Capítulos	Contributors
1	Empezando con estructuras de datos	Community, Keshav Sharma, Vardan
2	Apilar	Anup Kumar Gupta, Community, I. Afrin, Isha Agarwal, Jim Mischel, Vardan
3	Árbol de búsqueda binaria	Redet Getachew, venkatvb
4	Árbol de segmentos	Bakhtiar Hasan, Hesham Attia
5	Cola	Community, Isha Agarwal, lambda
6	Deque (doble cola de cola)	Isha Agarwal
7	Estructura de datos de búsqueda de la unión	Tobias Ribizel
8	Lista enlazada	Ashish Ahuja, Community, Ravindra HV
9	Matrices	Community
10	Montón binario	Vishwas
11	Recorridos de grafos	BPD1
12	Trie (árbol de prefijo / árbol de radix)	Bakhtiar Hasan, RamenChef