Estructuras de datos y algoritmos

Tutorías Ex. de Título 2024-2

Universidad Diego Portales
Prof. Víctor Reyes R.

Objetivos

- Complejidad algorítmica
- Estructuras de datos
- Algoritmos en grafos y árboles
- Algoritmos de ordenamiento

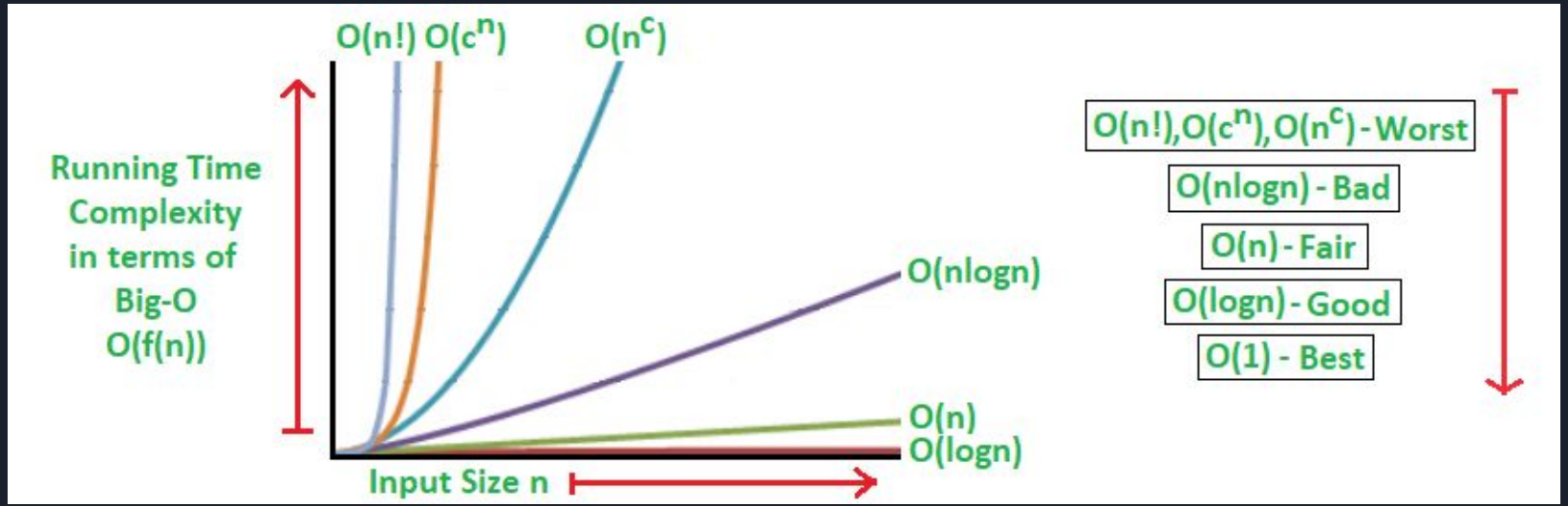
Análisis de complejidad

- ¿Para qué sirve?

  - Determina el tiempo y recursos requeridos para ejecutar un algoritmo
  - Sirve para comparar algoritmos dependientes del tamaño de entrada.
  - Ayuda para determinar la dificultad de un problema

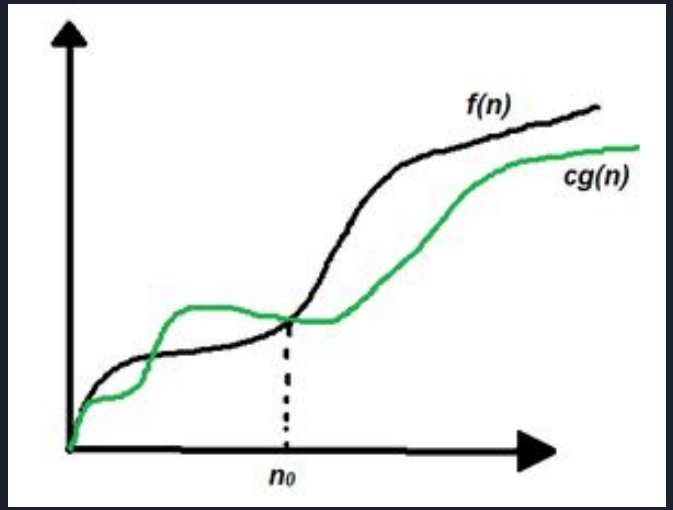
Notación asintótica: O

- Representa la cota superior del tiempo de ejecución de un algoritmo, es decir, peor caso de un algoritmo.



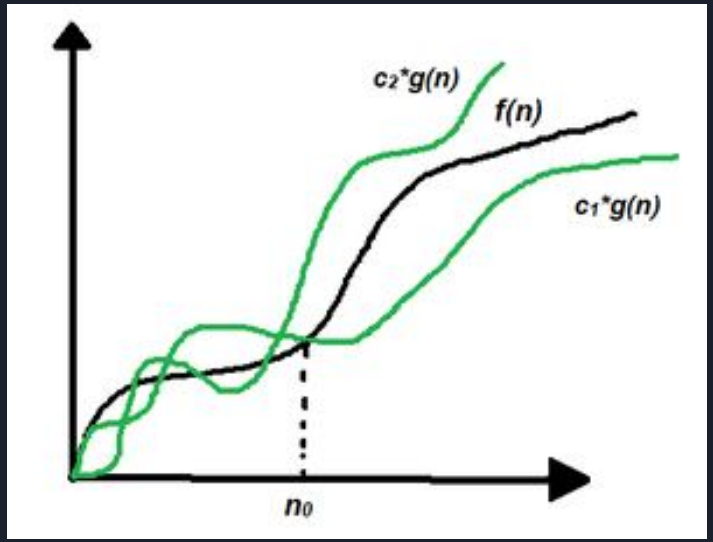
Notación asintótica: Ω

- La notación omega representa la cota inferior del tiempo de ejecución del algoritmo. Por tanto es el mejor caso posible.



Notación asintótica: Θ

- La notación theta representa tanto la cota inferior como superior, es decir el caso promedio de complejidad del algoritmo.



Notación asintótica: O

- Notaciones comunes de “big-O”:
  - O(1): tiempo constante
  - O(log n):tiempo logarítmico
  - O(n):tiempo lineal
  - O(nlog n): tiempo logarítmico lineal
  - O(n^2): tiempo cuadrático
  - O(n^3): tiempo cúbico
  - O(2^n): tiempo exponencial
  - O(n!): tiempo factorial

Notación asintótica O : Ejemplos

```
bool findElement(int arr[], int n, int key)
{
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            return true;
        }
    }
    return false;
}
```

Notación asintótica O : Ejemplos

```
bool findElement(int arr[], int n, int key)
{
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            return true;
        }
    }
    return false;
}
```

- Complejidad lineal

Notación asintótica O : Ejemplos

```
void multiply(int mat1[N][N], int mat2[N][N], int res[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            res[i][j] = 0;
            for (int k = 0; k < N; k++)
                res[i][j] += mat1[i][k] * mat2[k][j];
        }
    }
}
```

Notación asintótica O : Ejemplos

```
void multiply(int mat1[N][N], int mat2[N][N], int res[N][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            res[i][j] = 0;
            for (int k = 0; k < N; k++)
                res[i][j] += mat1[i][k] * mat2[k][j];
        }
    }
}
```

- Complejidad cúbica

Notación asintótica O : Ejemplos

```
int Alg(int arr[], int low, int high, int x)
{
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == x)
            return mid;
        if (arr[mid] < x)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}
```

Notación asintótica O : Ejemplos

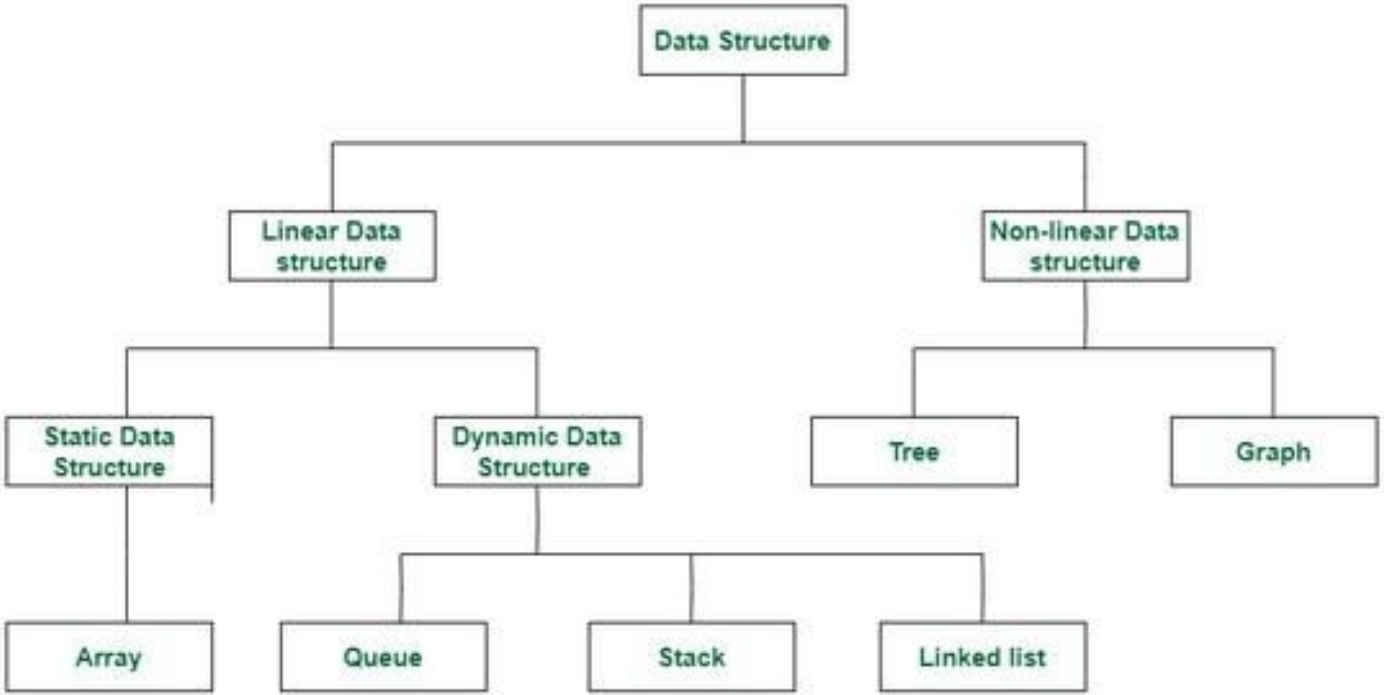
```
int binarySearch(int arr[], int low, int high, int x)
{
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == x)
            return mid;
        if (arr[mid] < x)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}
```

- Complejidad O(log n)

Estructuras de datos

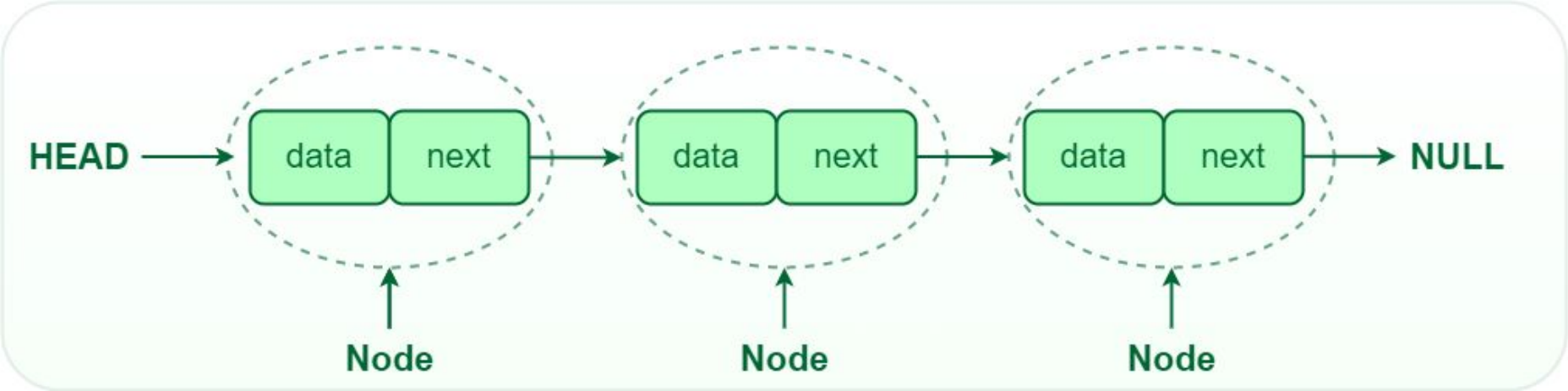
- Una estructura de datos es una forma organizada de almacenar y gestionar datos para facilitar su acceso y manipulación eficiente.

Classification of Data Structure



	Access	Search	Insertion	Deletion
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))
Splay Tree	-	Θ(log(n))	Θ(log(n))	Θ(log(n))
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))

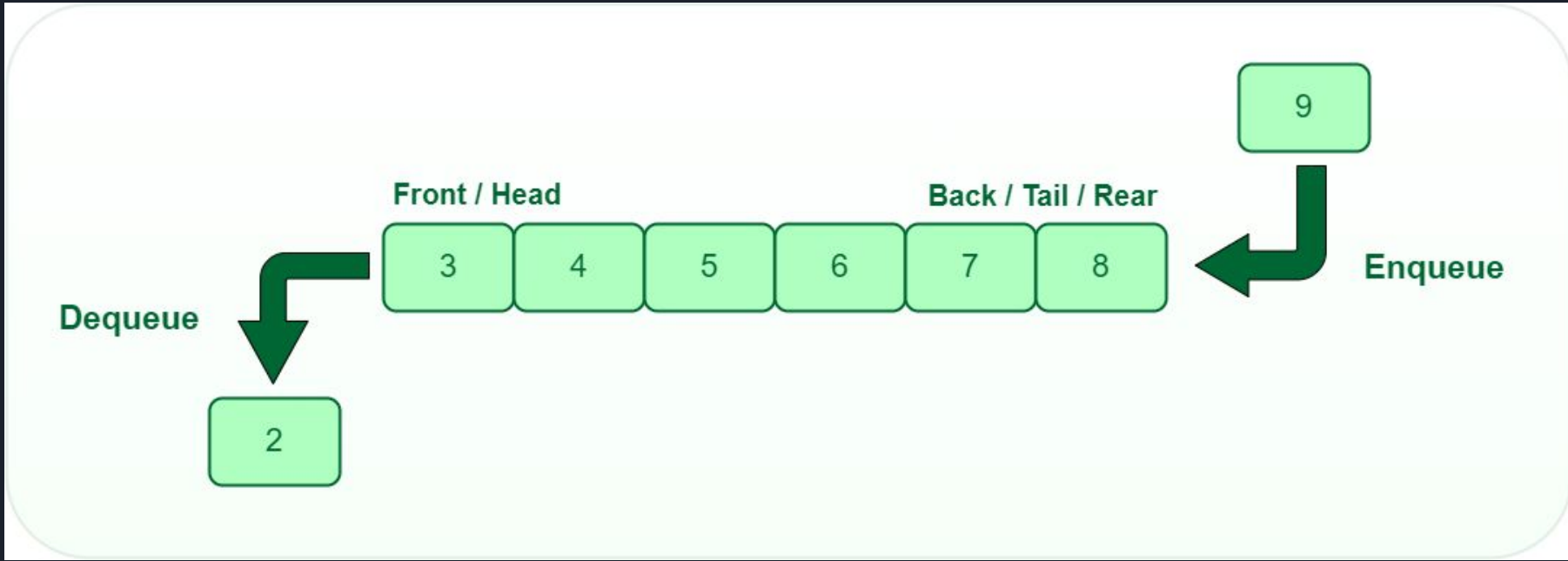
Lista enlazada



Lista enlazada

- Se puede acceder a la lista enlazada a través del nodo cabecera. El último nodo (nodo cola) apunta a NULL o nullptr, indicando el final de la lista.
- Eficientes para agregar y eliminar elementos (al principio de la lista), toman O(1) de tiempo.
- Acceso es peor que un arreglo: O(n) vs O(1)

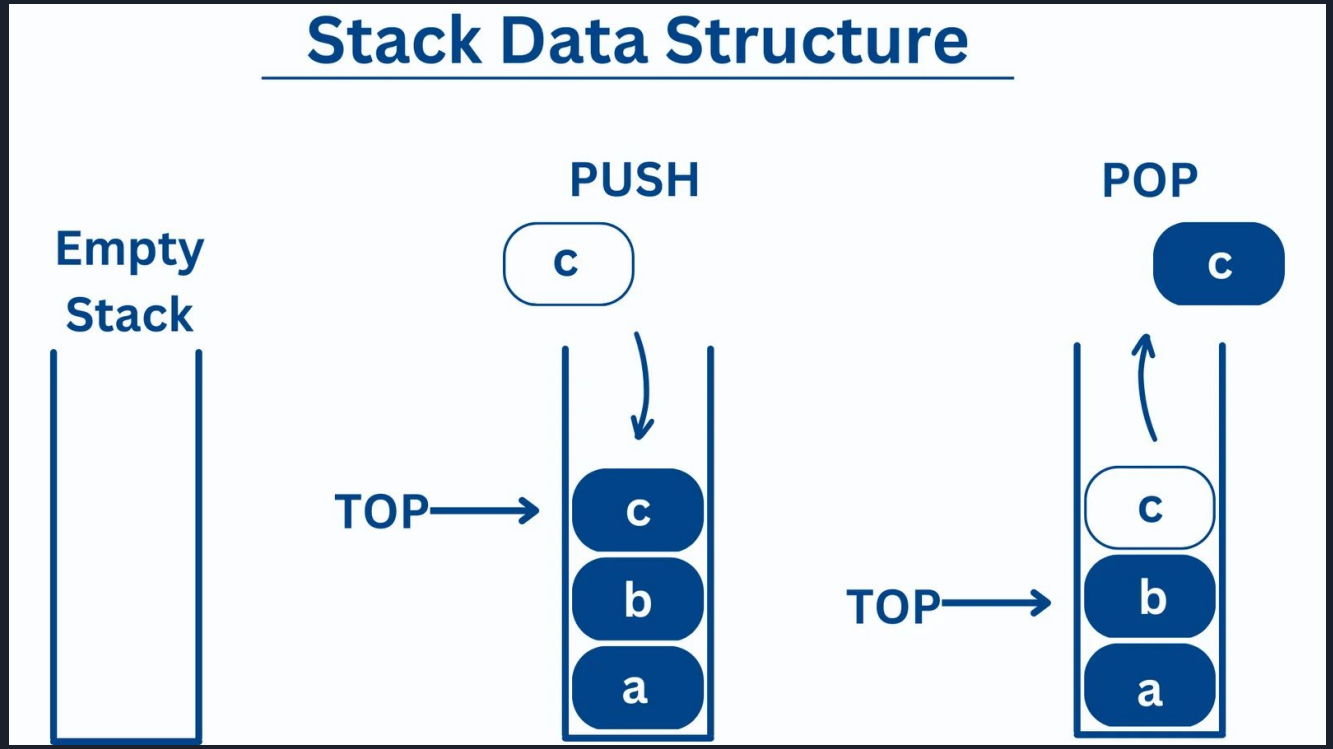
Queue



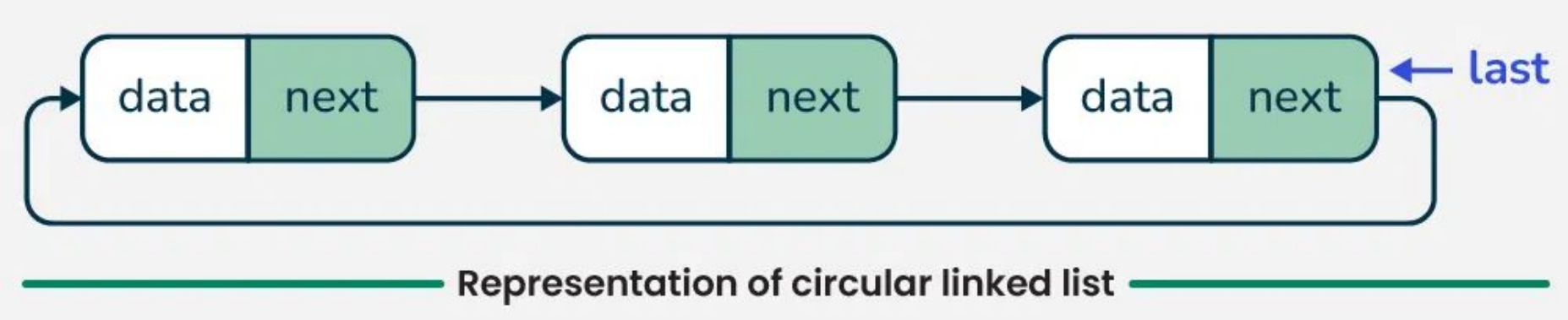
Queue

- Es del tipo FIFO (First in First Out)
- Aplicaciones más comunes: Recursos compartidos entre consumidores, información transferida de manera asincrónica entre dos procesos, entre otros.
- Operaciones más comunes: enqueue(x) O(1), dequeue O(1), front O(1)

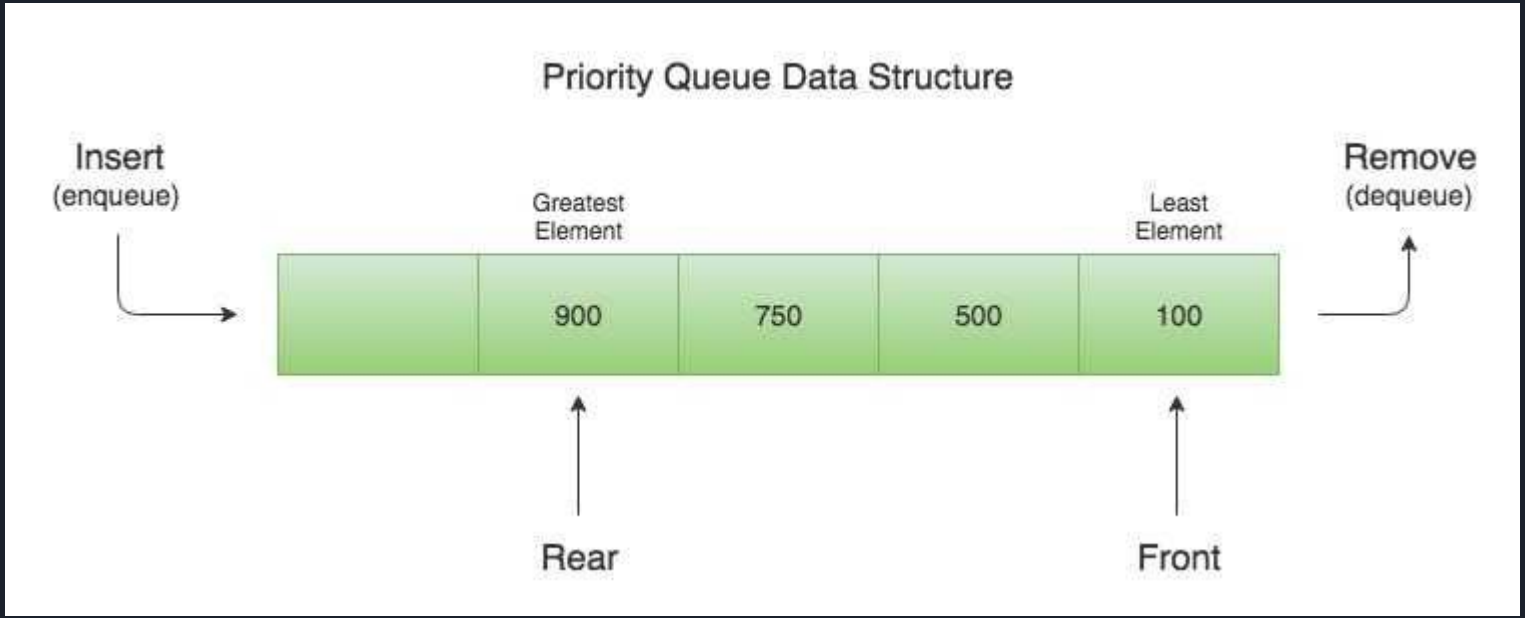
Stack



Lista circular



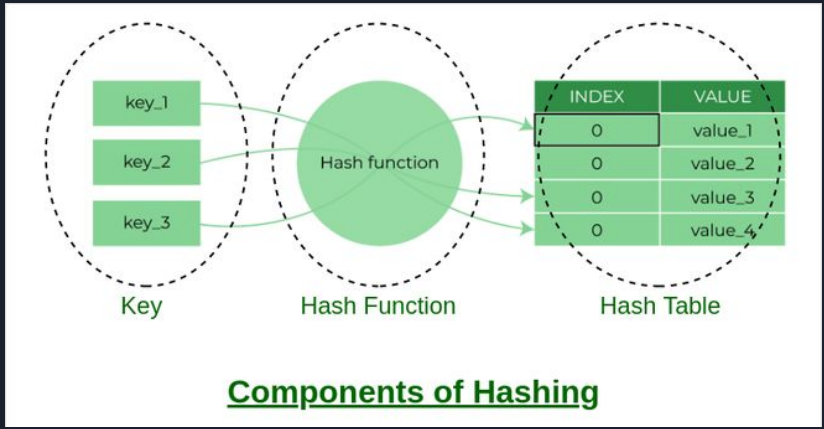
Priority queue



Tablas de hash

- Son diccionarios asociativos respecto de una llave y un dato vinculado.

- La asociación viene dada por una función de hash que recibe como parámetro el dato y entrega una llave.



Colisiones

- Una colisión en tablas hash ocurre cuando dos claves diferentes producen el mismo índice o dirección en la tabla después de pasar por la función hash.

- Las tablas hash tienen un tamaño fijo, por lo que el rango de valores que pueden producir es limitado. → Las funciones hash, aunque diseñadas para dispersar uniformemente las claves, no siempre pueden evitar que dos claves diferentes generen el mismo valor hash.

Colisiones

- Separate chaining/Encadenamiento: Cada posición de la tabla (índice) almacena una lista enlazada o una estructura similar para contener todas las claves que colisionan en esa posición.

- Open Addressing/Dirección abierta : En lugar de usar listas enlazadas, las colisiones se resuelven buscando la siguiente posición disponible en la tabla. →Lineal/Cuadrático/Hash doble

Grafos

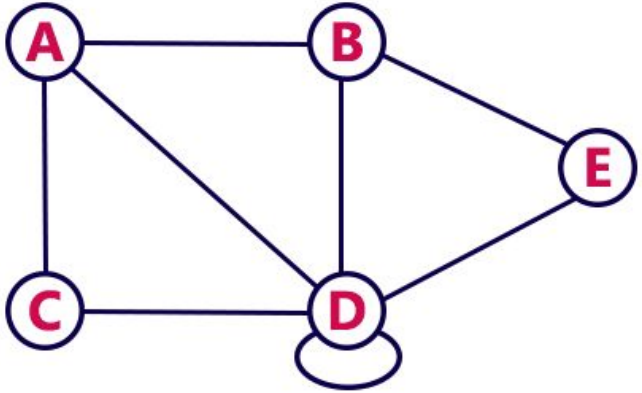
- Un grafo es una estructura de datos que surge de la conexión (aristas) entre vértices. Tiene variados usos y además tiene una cantidad enorme de algoritmos relacionados. Se denota generalmente como G = (V,E)

Grafos

- Los elementos básicos que están presentes en un grafo son:
  - Vértices: se refieren a aquellos puntos o nodos del grafo. Pueden contener datos o no.
  - Aristas: son las conexiones que comunican a un par de vértices. Pueden tener valores asociados (como costos) o no. Asimismo, las aristas pueden tener direccionalidad.

Grafos: Representación

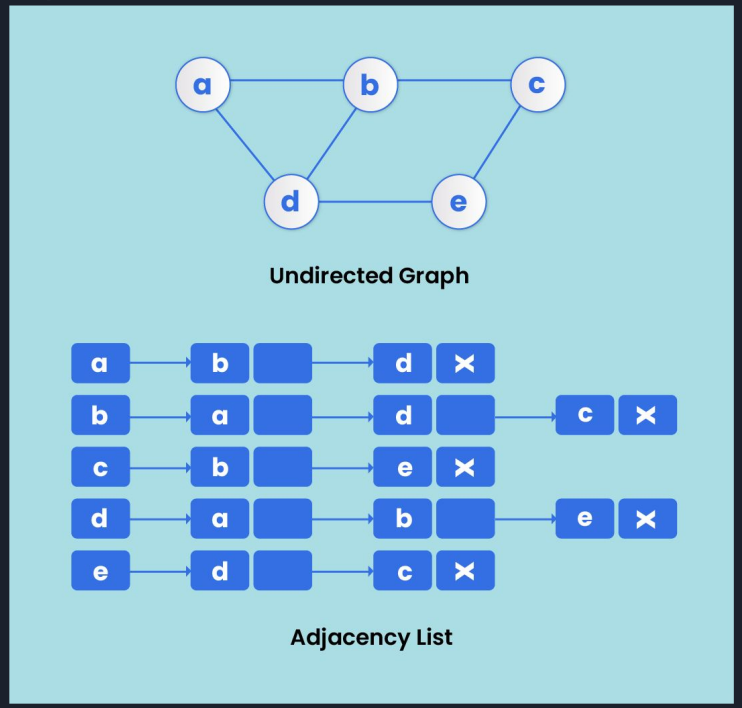
- Gráfico / Matriz de adyacencia



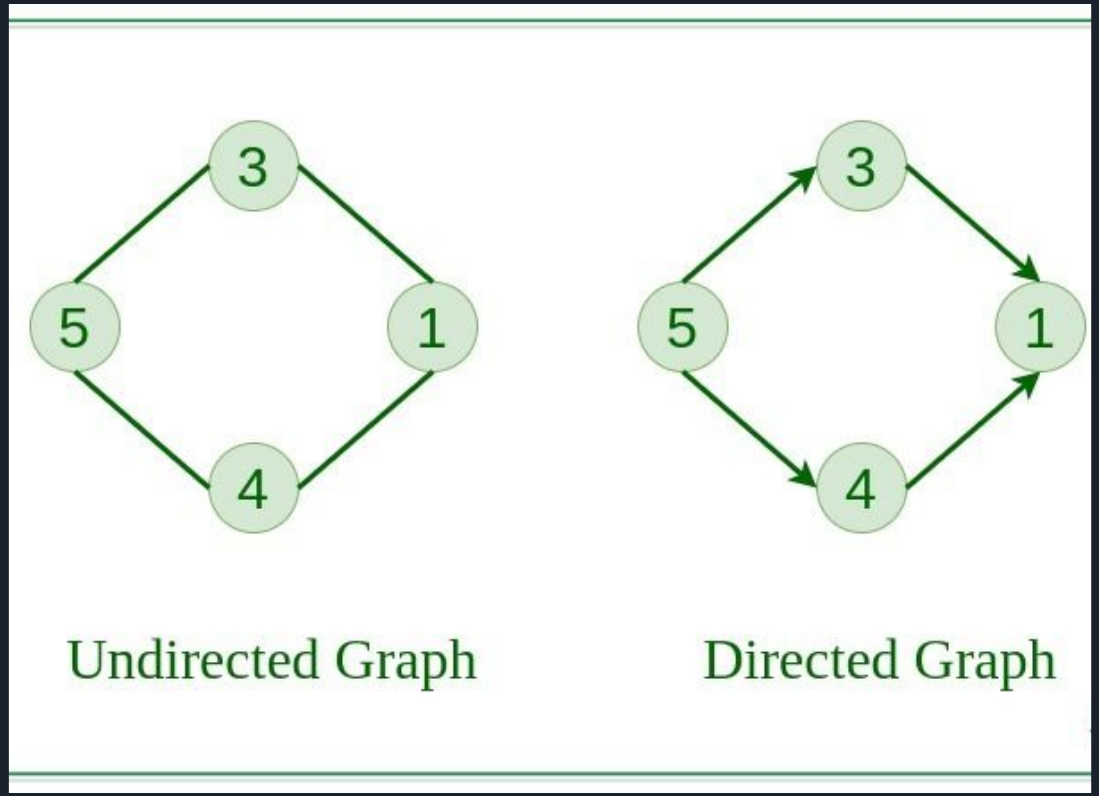
	A	B	C	D	E
A	0	1	1	1	0
B	1	0	0	1	1
C	1	0	0	1	0
D	1	1	1	1	1
E	0	1	0	1	0

Grafos: Representación

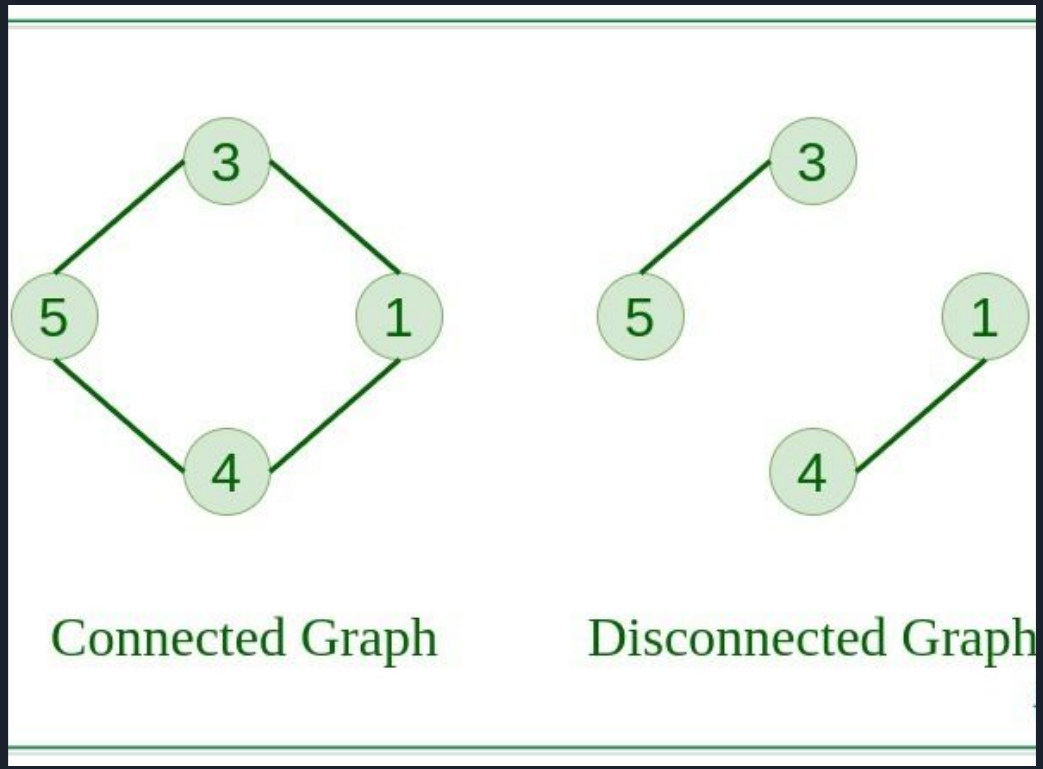
- Gráfico / Lista de adyacencia



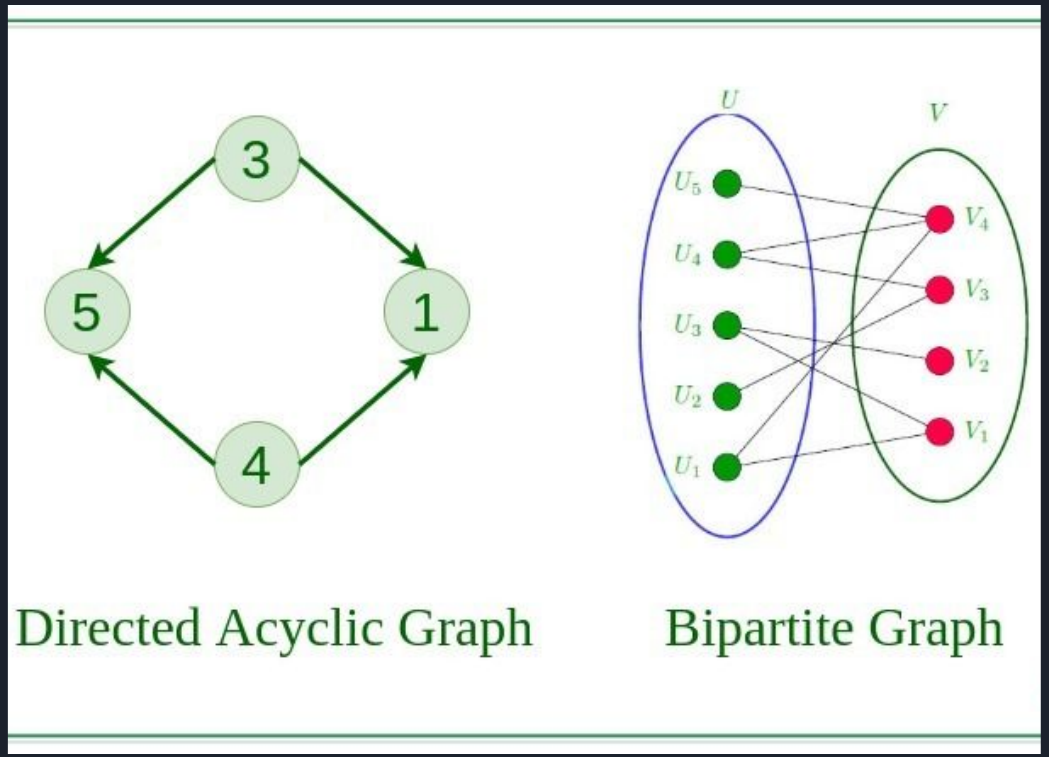
Ejemplos de grafos



Eiemplos de grafos



Ejemplos de grafos



Grafos: Aplicaciones

- Como se mencionó previamente, las aplicaciones que tienen los grafos son extremadamente variadas y amplias.

- Los grafos se pueden usar como mapas que establecen localizaciones y rutas que las conectan (esto se puede asociar a transporte, aunque también tiene que ver con diseño de redes).

Algoritmos

- Revisaremos tres algoritmos en esta sesión:
  - Prim y Kruskal: Orientados a encontrar árboles de cobertura mínimos.
  - Dijkstra: Sirve para encontrar las rutas de menor costo entre vértices de grafos.

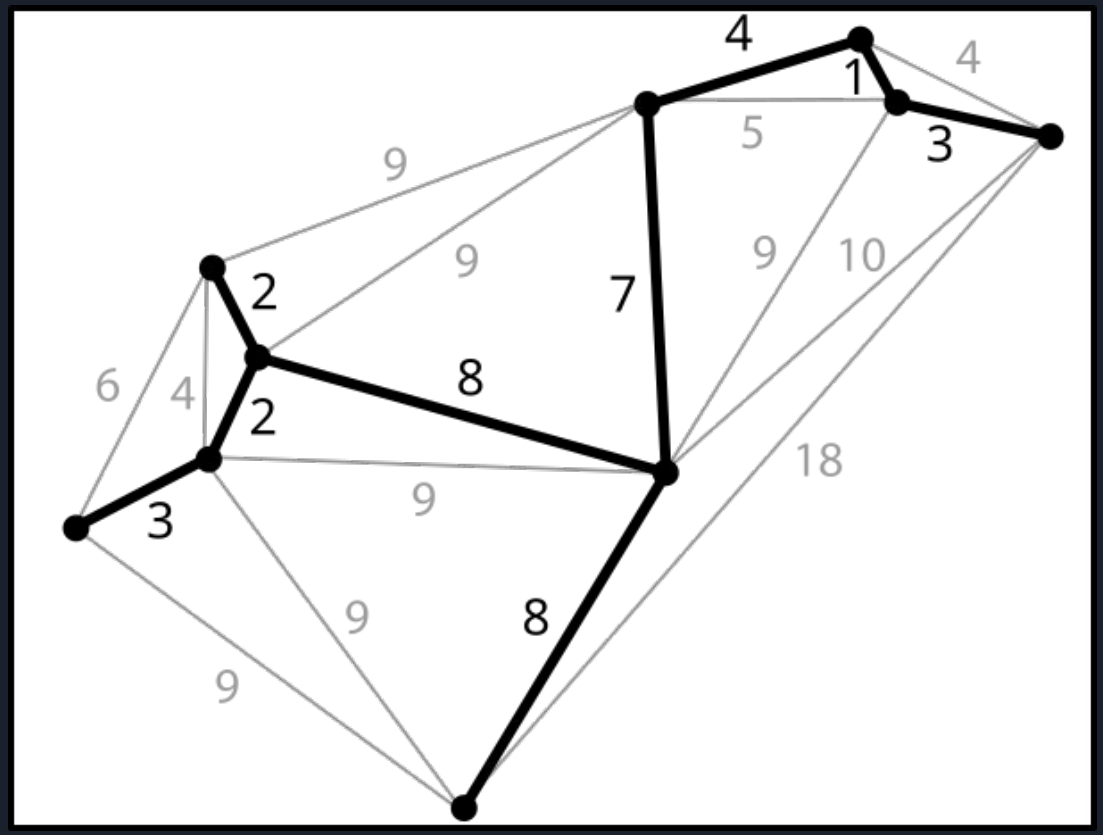
Árbol de cobertura mínimo

- Un árbol de cobertura está formado por los vértices de un grafo y un subconjunto de sus aristas. Es una configuración acíclica que permite transitar de cualquier vértice a cualquier otro en el grafo.

- Se debe notar que estos árboles se construyen sobre grafos no dirigidos y que tienen costos asociados a sus aristas. Asimismo, se tratan de grafos conexos.

- Revisaremos dos técnicas: Prim y Kruskal (ambos greedy)

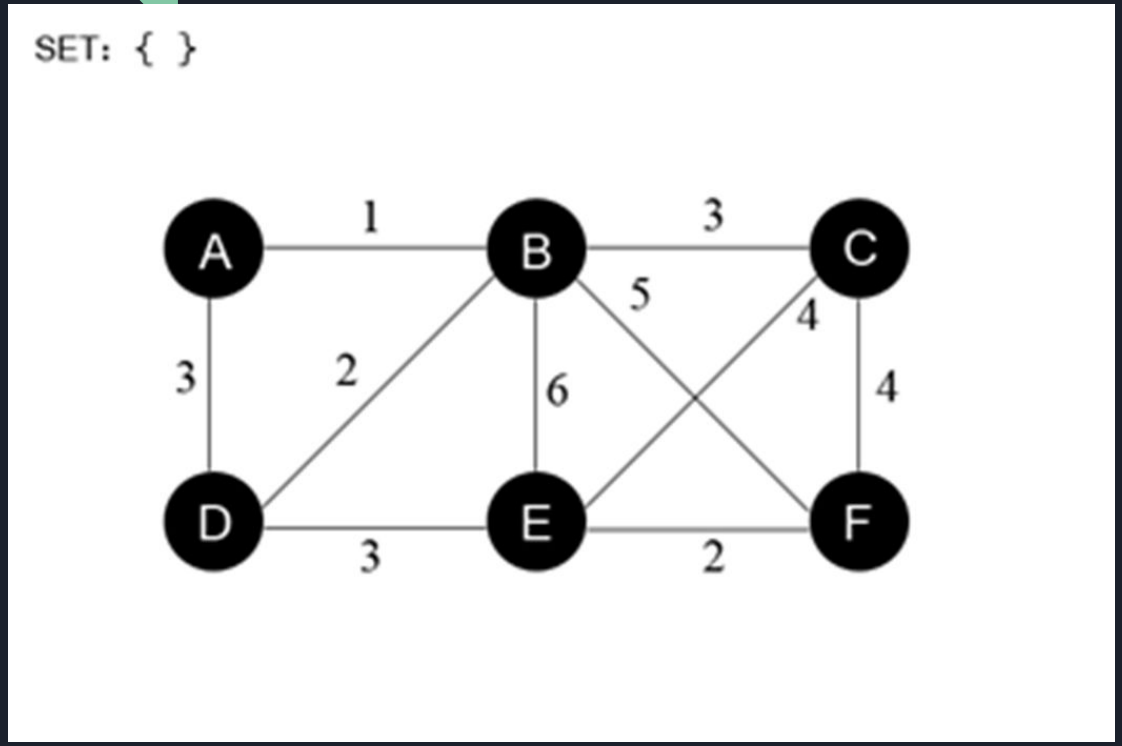
Árbol de cobertura mínimo



Algoritmo de Prim

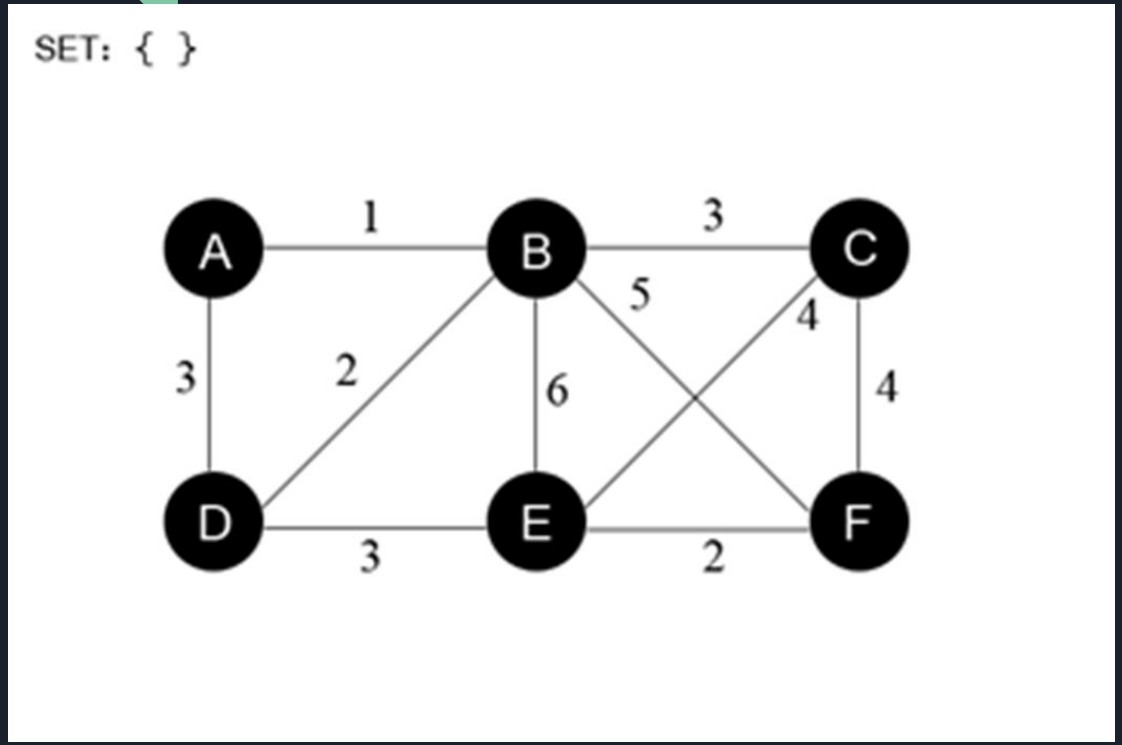
- A partir de un grafo G:
  - Seleccionar un vértice v_1 al azar
  - Incluirlo en el conjunto de vértices visitados (ST)
  - Mientras no se hayan visitado todos los vértices de V:
    - Encontrar el conjunto de las aristas que conectan a algún vértice en ST con uno no-visitado (v_2)
    - Elegir la arista de menor costo e incluir a v_2 en ST

Algoritmo de Prim: Ejemplo



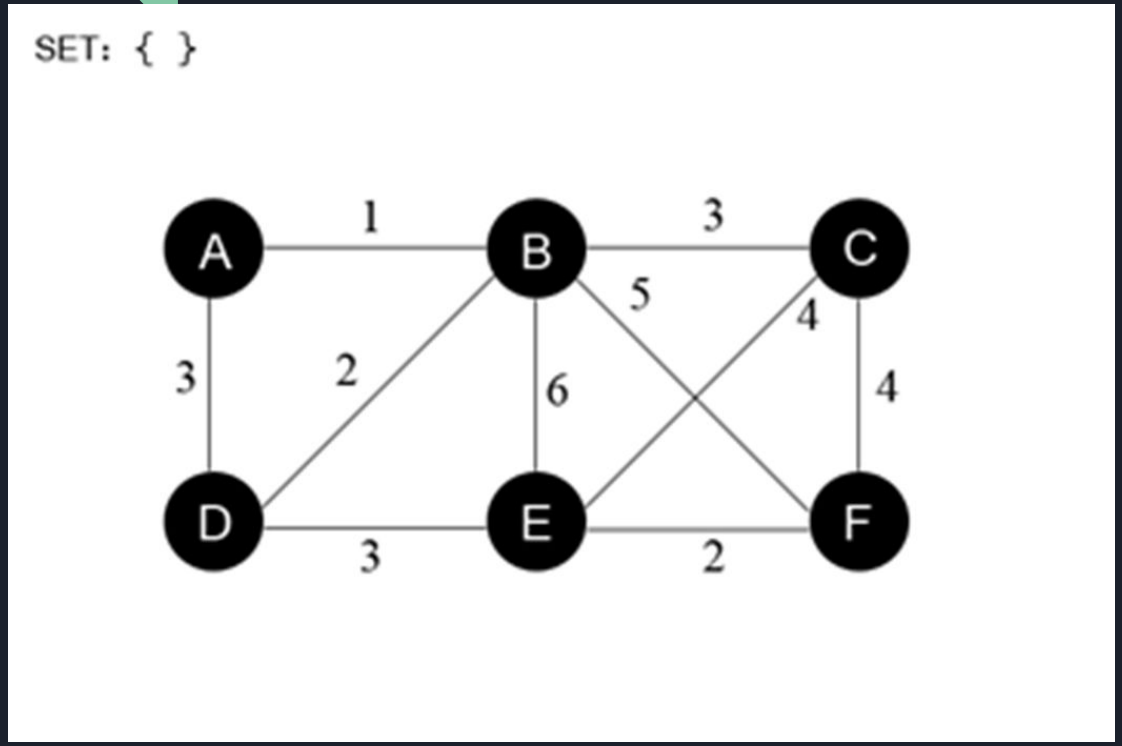
- Para cada nodo, inicializamos su costo en infinito y su vecino en null.
- Elegimos un nodo aleatorio, por ejemplo E y su costo queda en 0
- Actualizamos el costo de D a 3 y su vecino a E, costo de B a 6 y vecino E, F costo de 2 y vecino E, costo de C a 4 y vecino E.

Algoritmo de Prim: Ejemplo



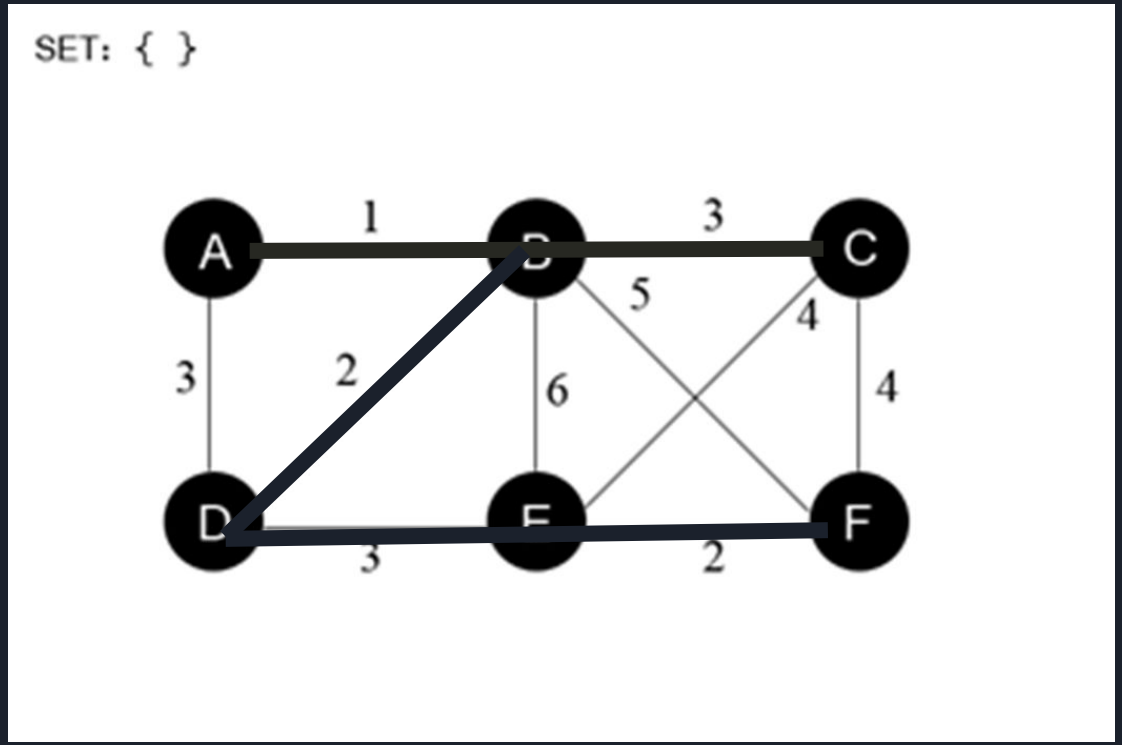
- Set={E}. Elegimos el de menor costo, osea F. Actualizamos el costo de B a 5 y su vecino a F (es mejor que E)
- Set queda en {E,F}

Algoritmo de Prim: Ejemplo



- Menor costo D. Actualizamos el costo de A a 3 y el nodo D como vecino, y también actualizamos el costo de B a 2 y a D como su vecino.
- SET={E,F,D}
- El de menor costo es B. Actualizamos A a costo 1 y su vecino B, y C a costo 3 y vecino B.
- SET={E,F,D,B} …seguir

Algoritmo de Prim: Ejemplo

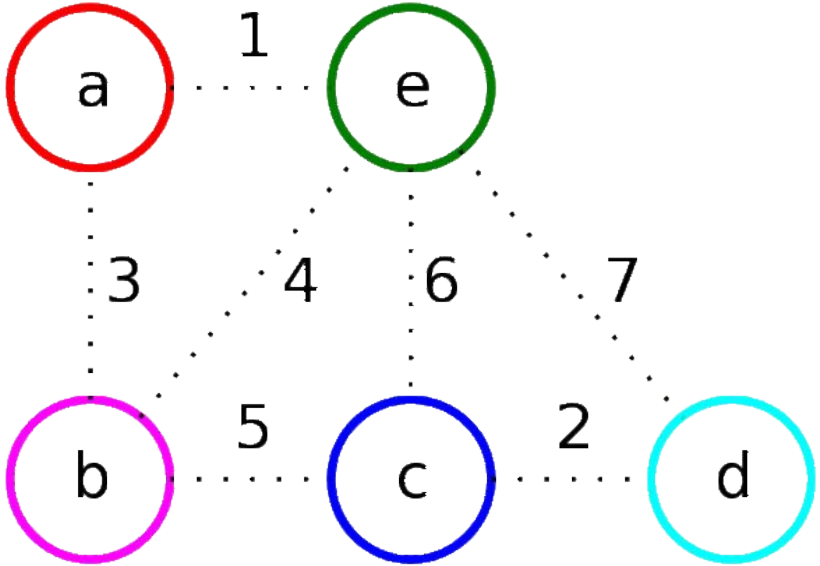


Algoritmo de Kruskal

- A partir de un grafo G:
  - Ordenar el conjunto de las aristas E por costo
  - Seleccionar e incluir la arista de menor costo en el conjunto (e_1) de aristas presentes (ST) y registrar los vértices alcanzados
  - Mientras no se hayan alcanzado todos los vértices de V:
    - Seleccionar la siguiente arista de menor costo que no forme un ciclo (e_2) con las aristas en ST
    - Agregar e_2 a ST

Algoritmo de Kruskal

Edge	ab	ae	bc	be	cd	ed	ec
Weight	3	1	5	4	2	7	6



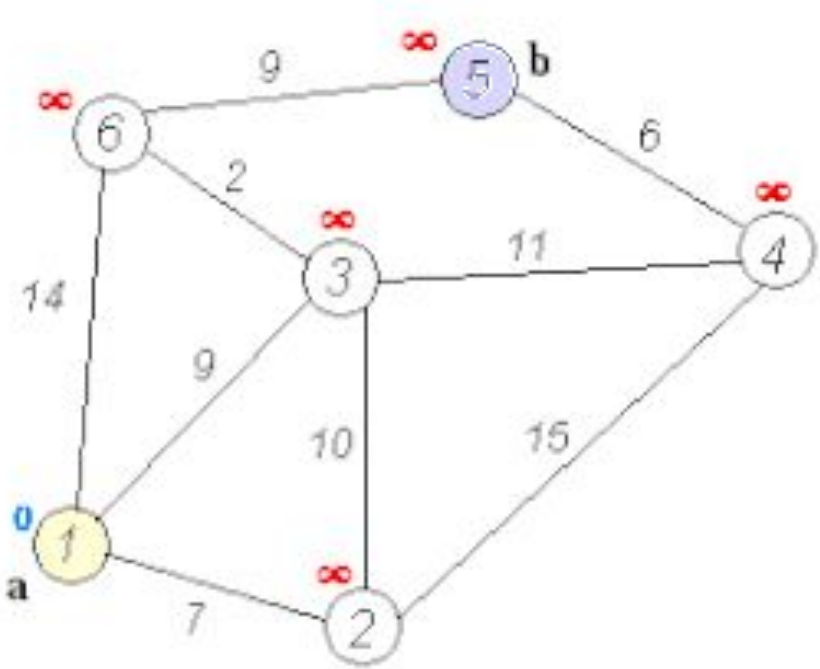
Algoritmo de Dijkstra

- Calcula los costos de todas las rutas mínimas desde un vértice de origen a todos los otros vértices en un grafo G.
- No admite aristas de costo negativo.
- Trabaja sobre grafos dirigidos.

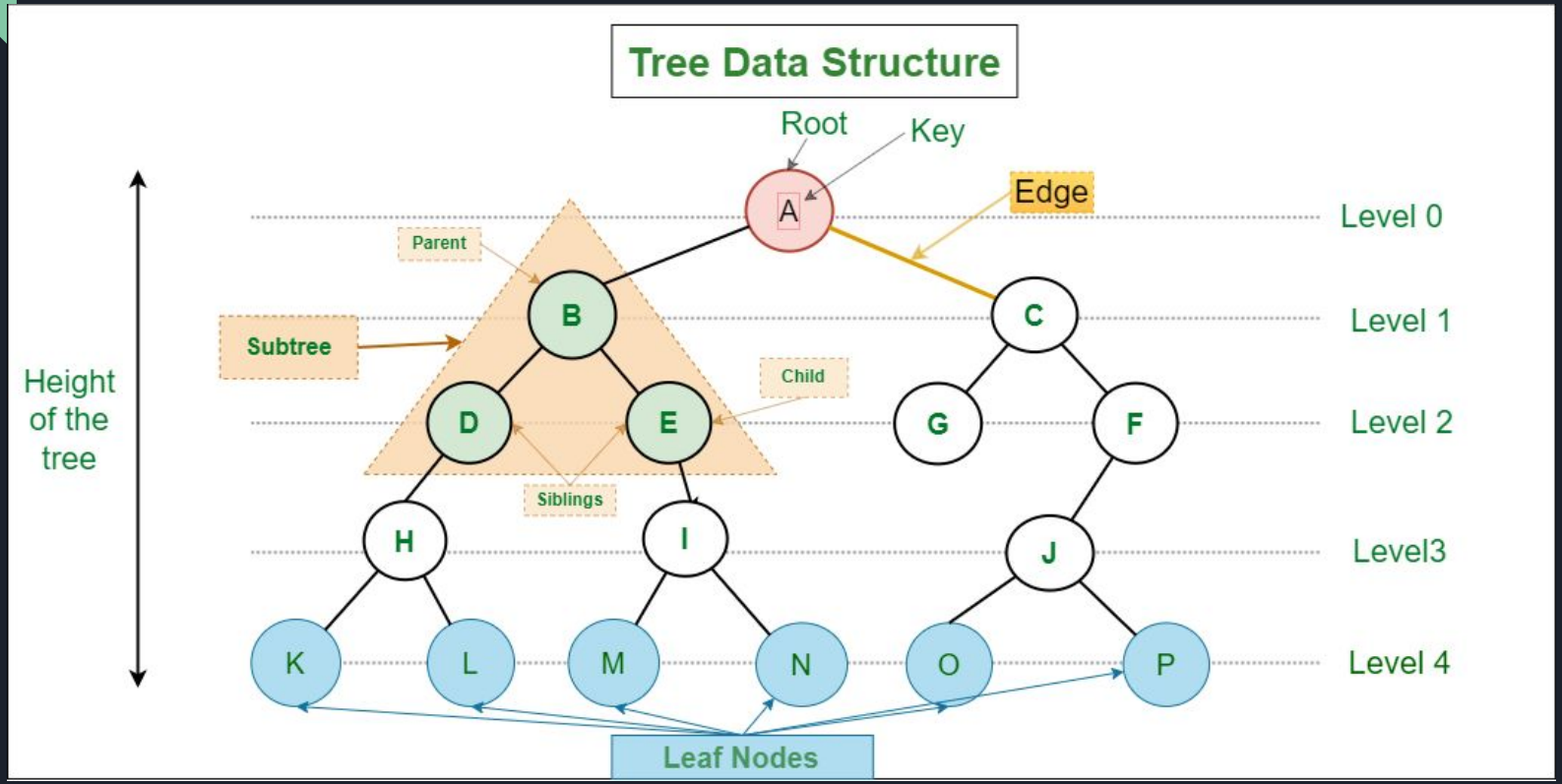
Algoritmo de Dijkstra

- A partir de un grafo G:
  - Se define la matriz de adyacencia (D^0).
  - Se crea un vector de costos C de cardinalidad igual a V.
  - Se inicializa este vector con todos los valores en ∞.
  - Se crea un vector (inicialmente vacío) de vértices visitados W.
  - Se selecciona el vértice de origen O y se coloca C(O) = 0.
  - Mientras no se hayan visitado todos los vértices:
    - Seleccionar el vértice S que no esté en W de menor costo (C(S)) a algún vértice en W.
    - Para cada vértice N vecino de S, $C(N) = \min(C(N), C(S) + D^0(S,N))$.
    - Agregar S a W.

Algoritmo de Dijkstra



Árboles


Tree Data Structure
Root
Key
Edge
A
Parent
B
C
Subtree
Child
D
E
G
F
Siblings
H
I
J
K
L
M
N
O
P
Height of the tree
Level 0
Level 1
Level 2
Level3
Level 4
Leaf Nodes

Búsqueda en anchura

- Algoritmo utilizado para explorar o buscar un nodo en un grafo o árbol de manera sistemática.

- Se caracteriza por expandir primero los nodos más cercanos al nodo inicial antes de explorar los nodos más profundos. Es decir, la BFS examina todos los nodos de un nivel antes de pasar al siguiente nivel.

Búsqueda en anchura

- BFS(Árbol A, Nodo raíz):
  - Crear una cola Q
  - Agregar el nodo inicial a la cola Q
  - Marcar el nodo inicial como visitado
  - Mientras Q no esté vacía:
    - nodo_actual = Q.desencolar()
    - Procesar nodo_actual
    - Para cada nodo vecino en nodo_actual:
      - Si el vecino no ha sido visitado: Marcar el vecino como visitado
      - Encolar el vecino en Q

Búsqueda en profundidad

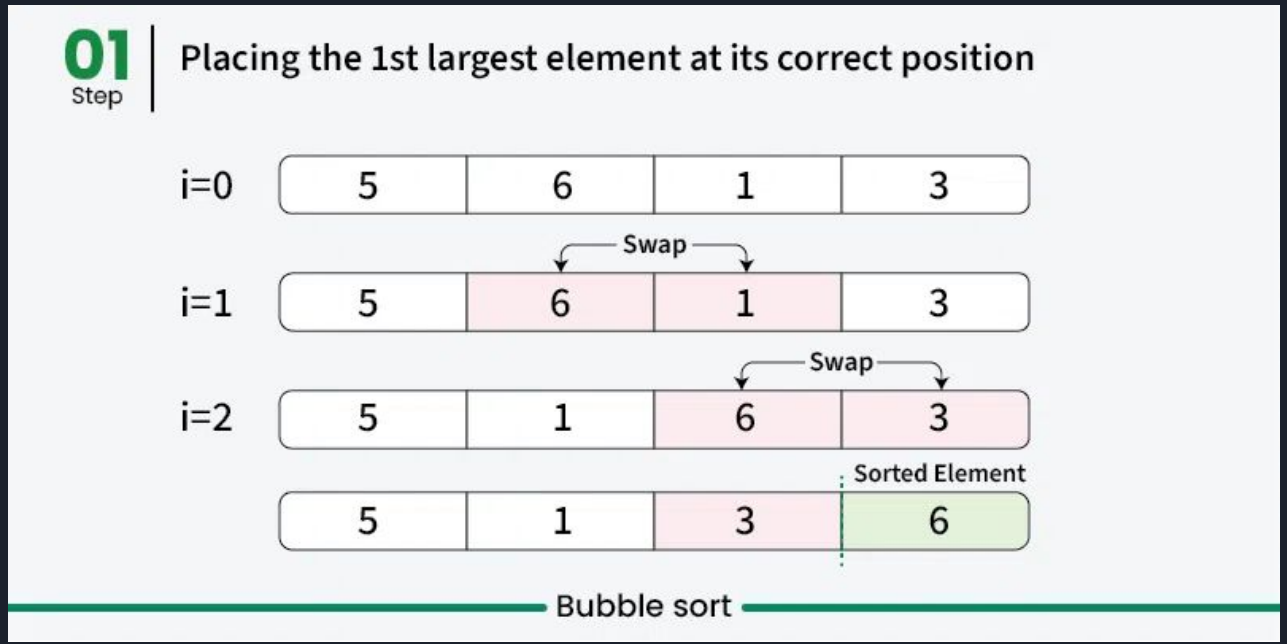
- Algoritmo utilizado para explorar o buscar un nodo en un grafo o árbol de manera sistemática.
- En un árbol, la DFS explora tan profundo como sea posible por cada rama antes de retroceder y explorar las ramas alternativas.
- Intenta alcanzar un nodo final en una rama antes de considerar otras opciones.

Búsqueda en profundidad

- DFS(Árbol A, Nodo inicial):
  - Crear una pila S
  - Apilar el nodo inicial en S
  - Mientras S no esté vacía:
    - nodo_actual = S.desapilar()
    - Si nodo_actual no ha sido visitado: Marcar nodo_actual como visitado
    - Procesar nodo_actual
    - Apilar cada hijo de nodo_actual en S (en orden inverso)

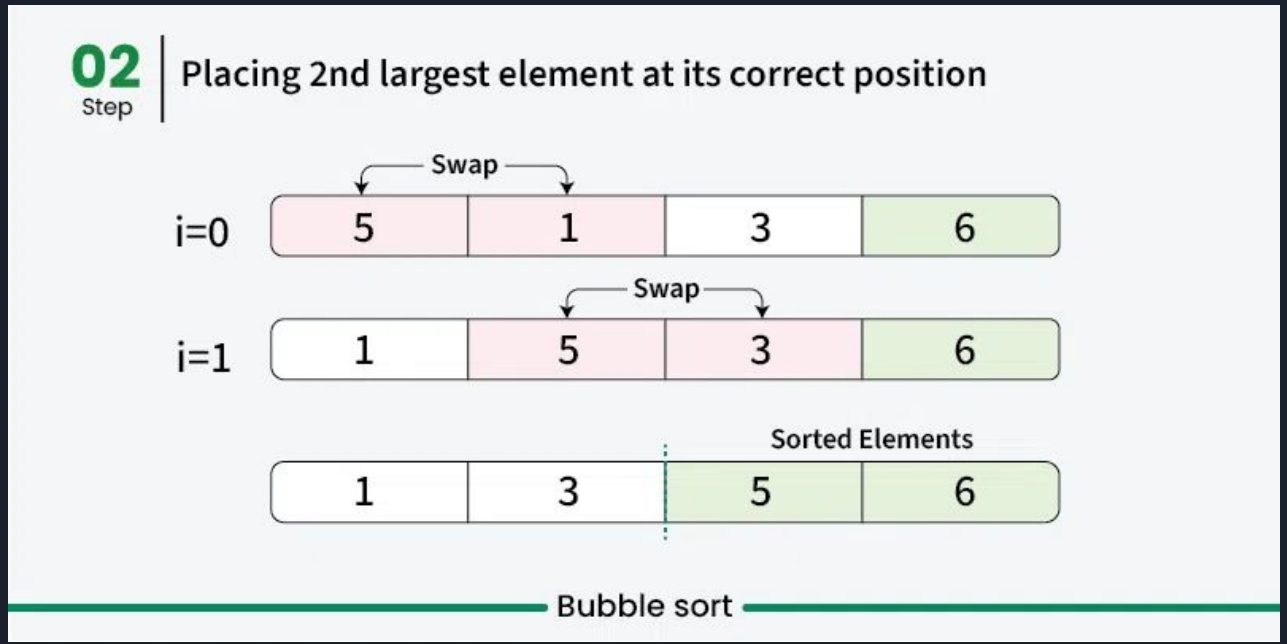
Algoritmos de Ordenamiento

- Bubble Sort



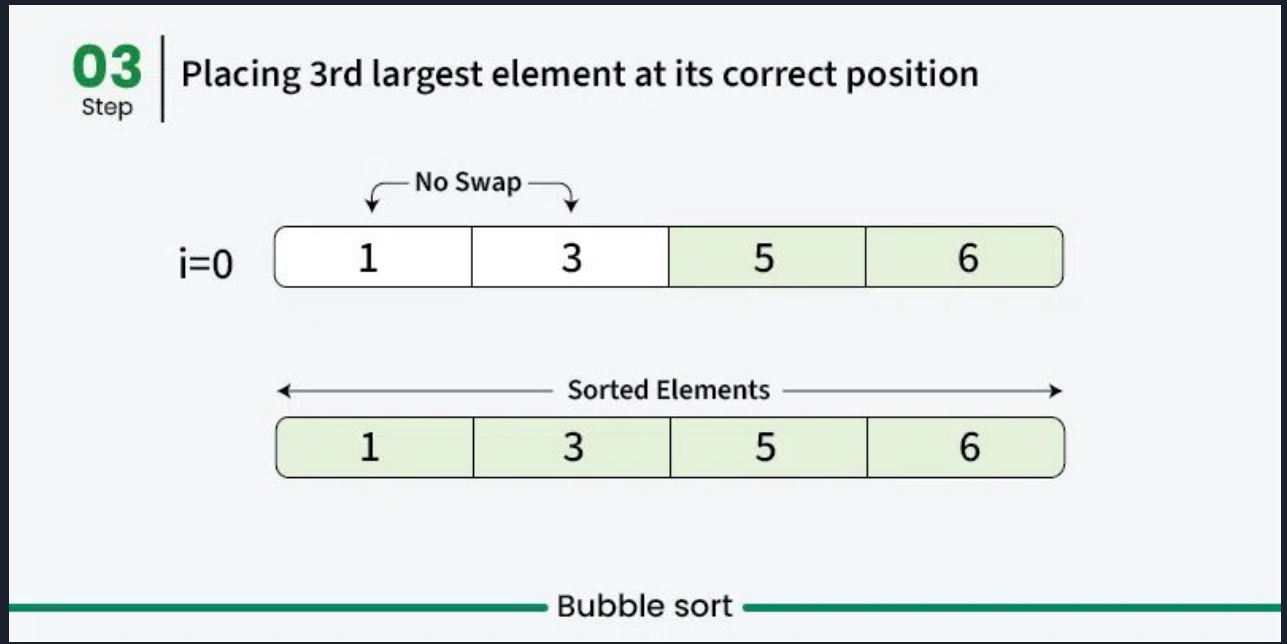
Algoritmos de Ordenamiento

- Bubble Sort



Algoritmos de Ordenamiento

- Bubble Sort



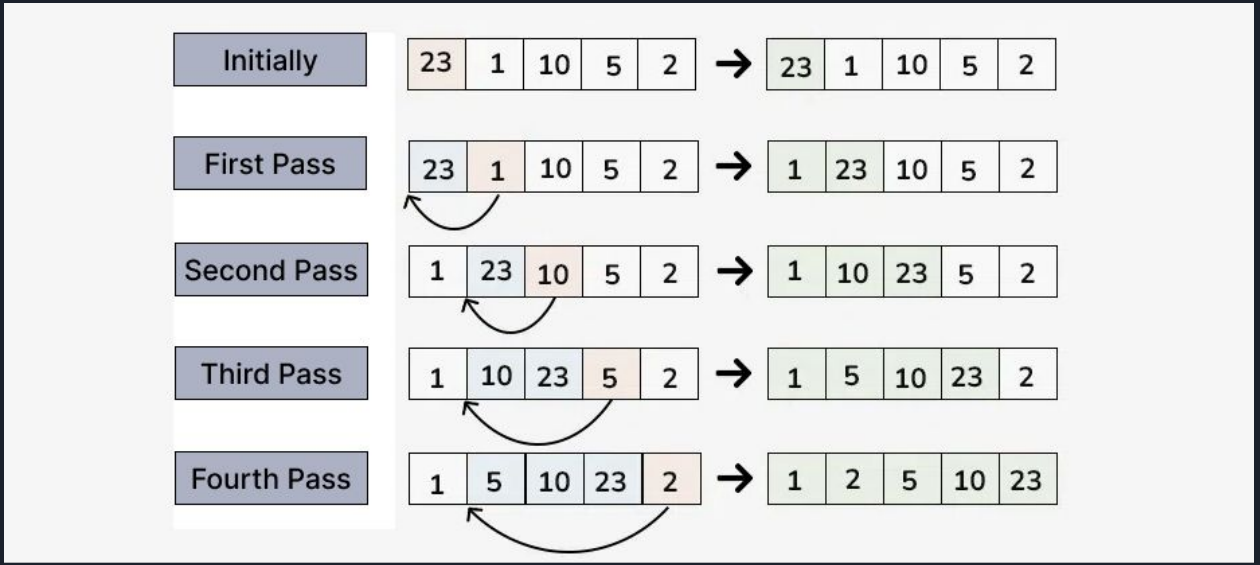
Algoritmos de Ordenamiento

- Bubble Sort → O(n^2)

```cpp
void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    bool swapped;

    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }

        // If no two elements were swapped, then break
        if (!swapped)
            break;
    }
}
```

Algoritmos de Ordenamiento

- Insertion Sort



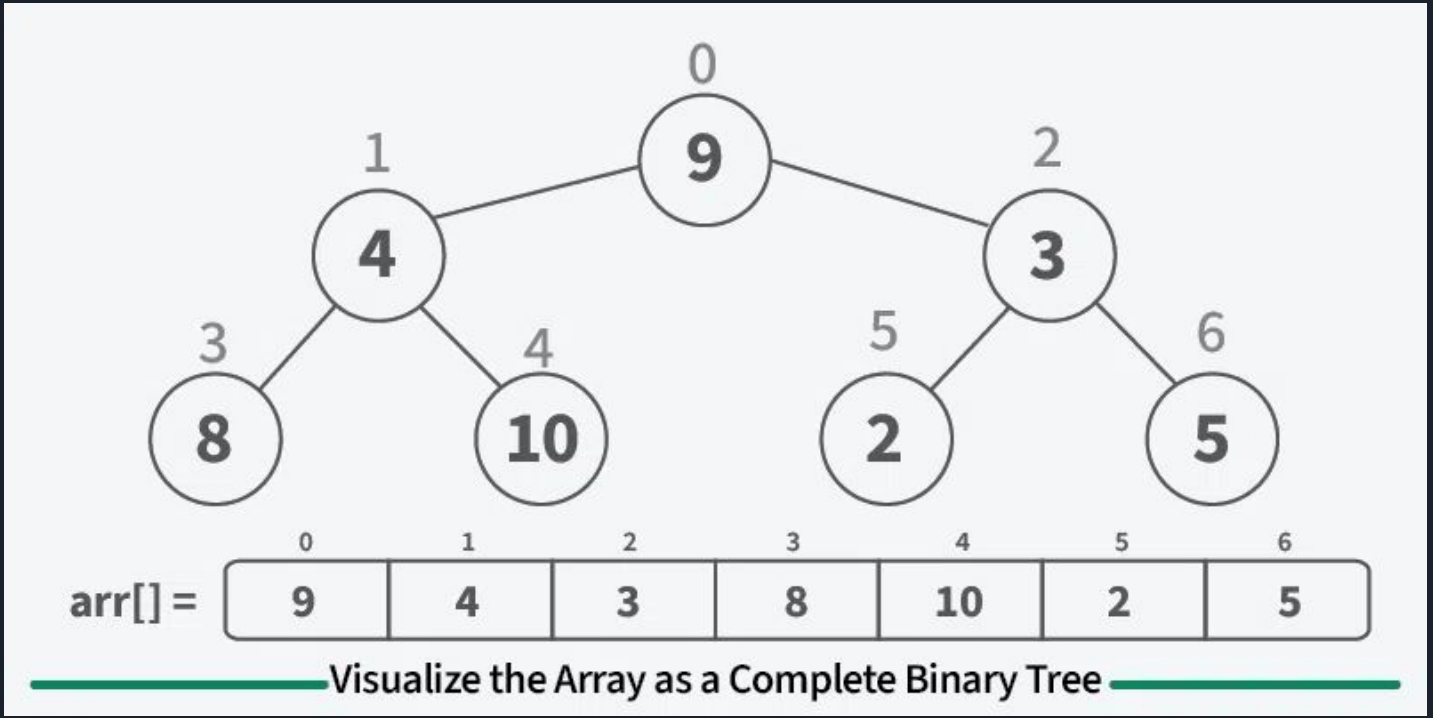
Algoritmos de Ordenamiento

- Insertion Sort → O(n^2)

```
void insertionSort(int arr[], int n)
{
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        /* Move elements of arr[0..i-1], that are
           greater than key, to one position ahead
           of their current position */
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j = j - 1;
        }
        arr[j + 1] = key;
    }
}
```

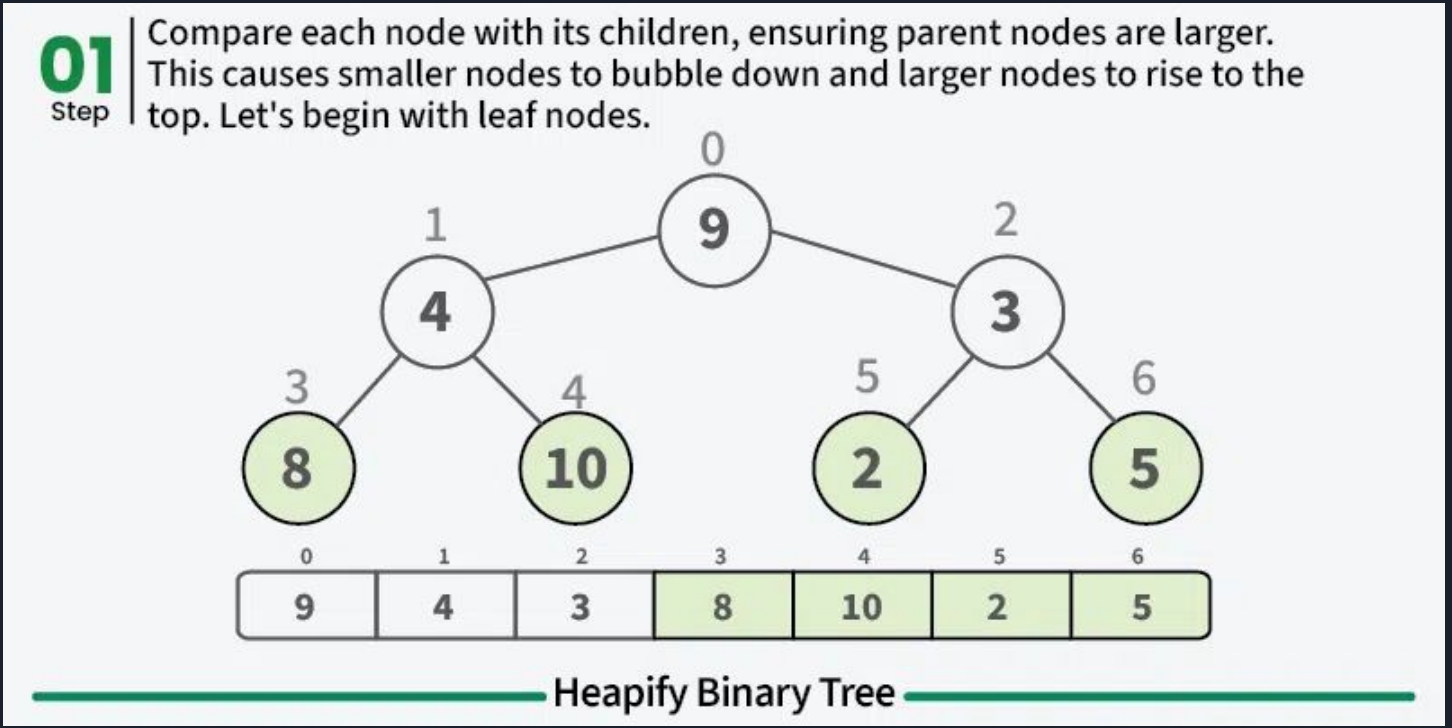
Algoritmos de Ordenamiento

- Heap Sort



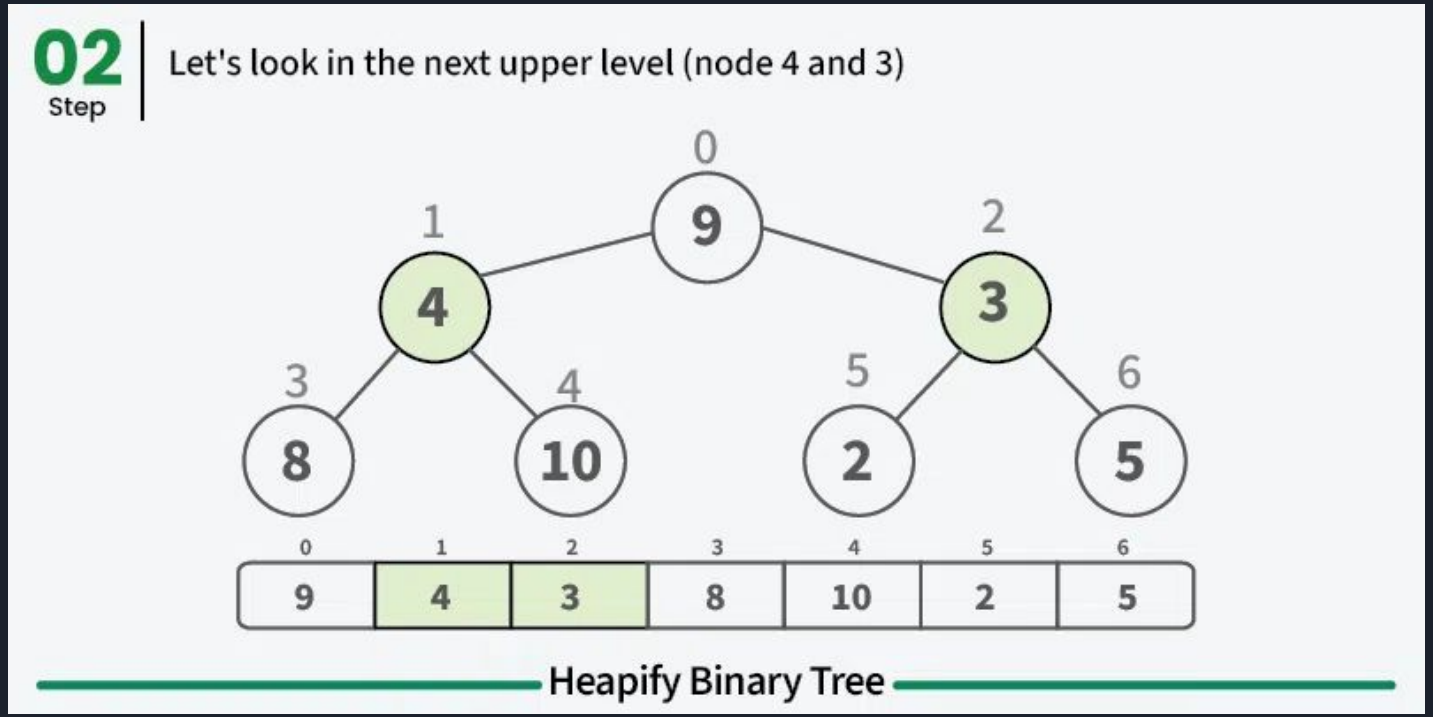
Algoritmos de Ordenamiento

- Heap Sort



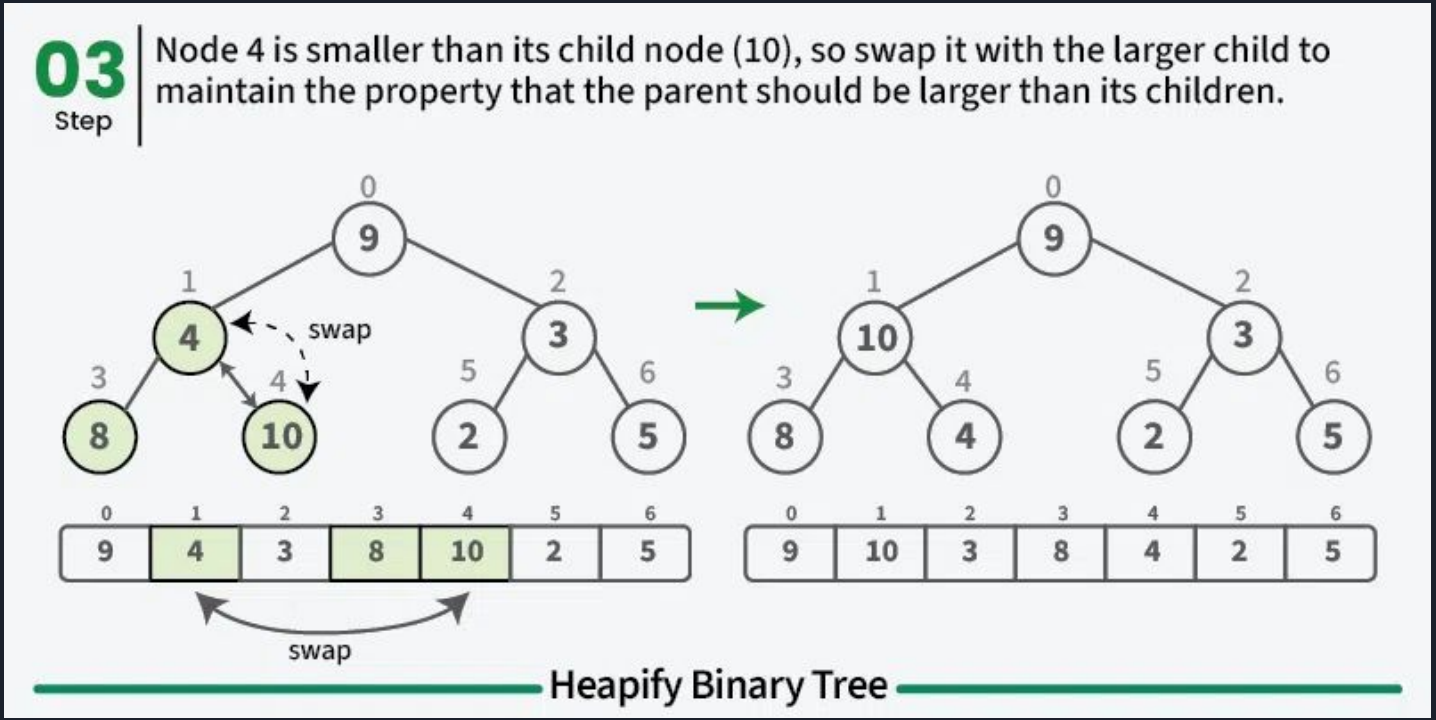
Algoritmos de Ordenamiento

- Heap Sort



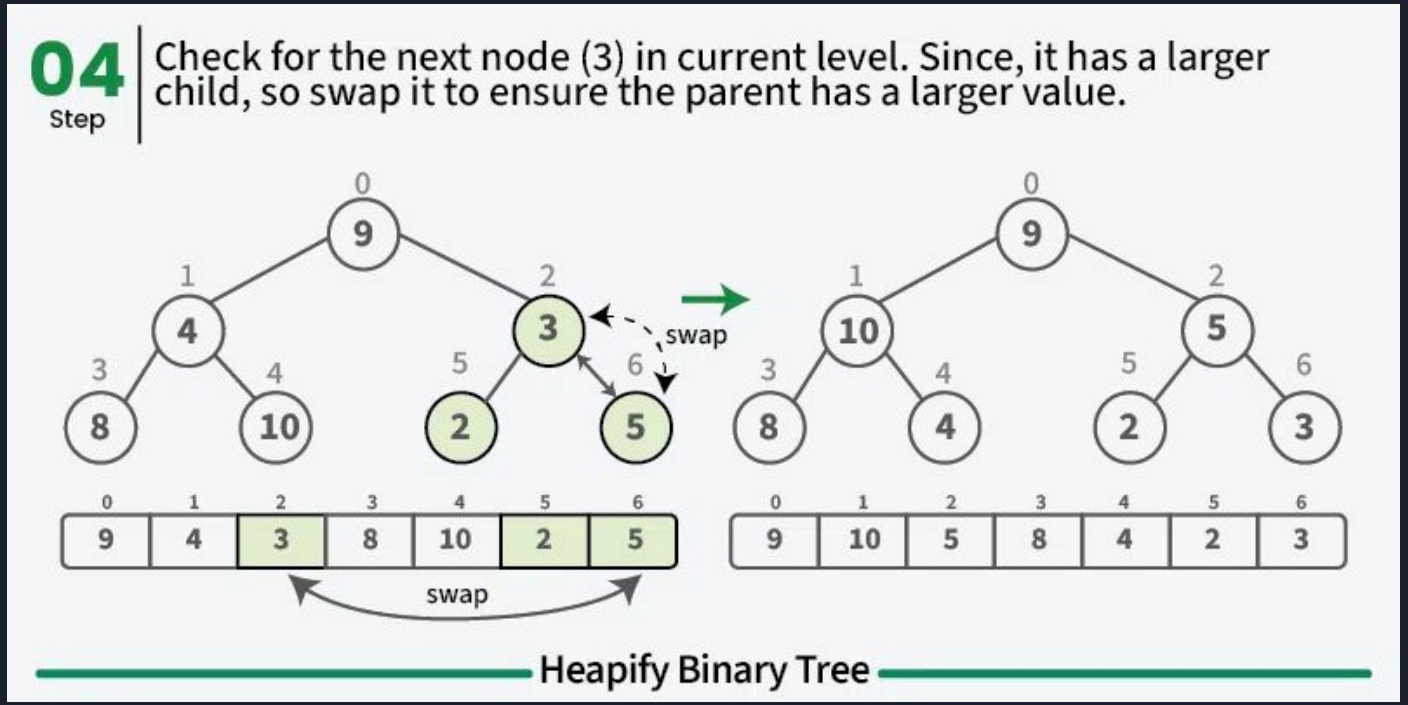
Algoritmos de Ordenamiento

- Heap Sort



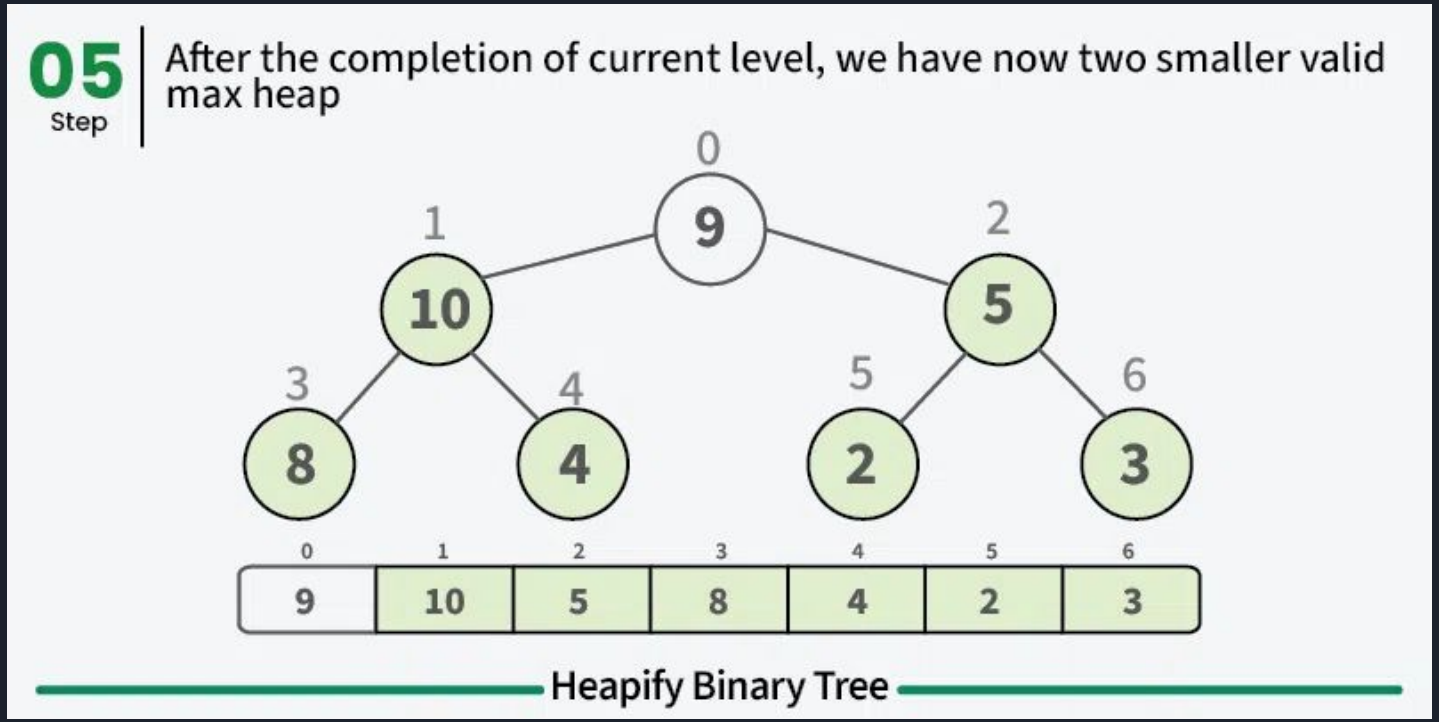
Algoritmos de Ordenamiento

- Heap Sort



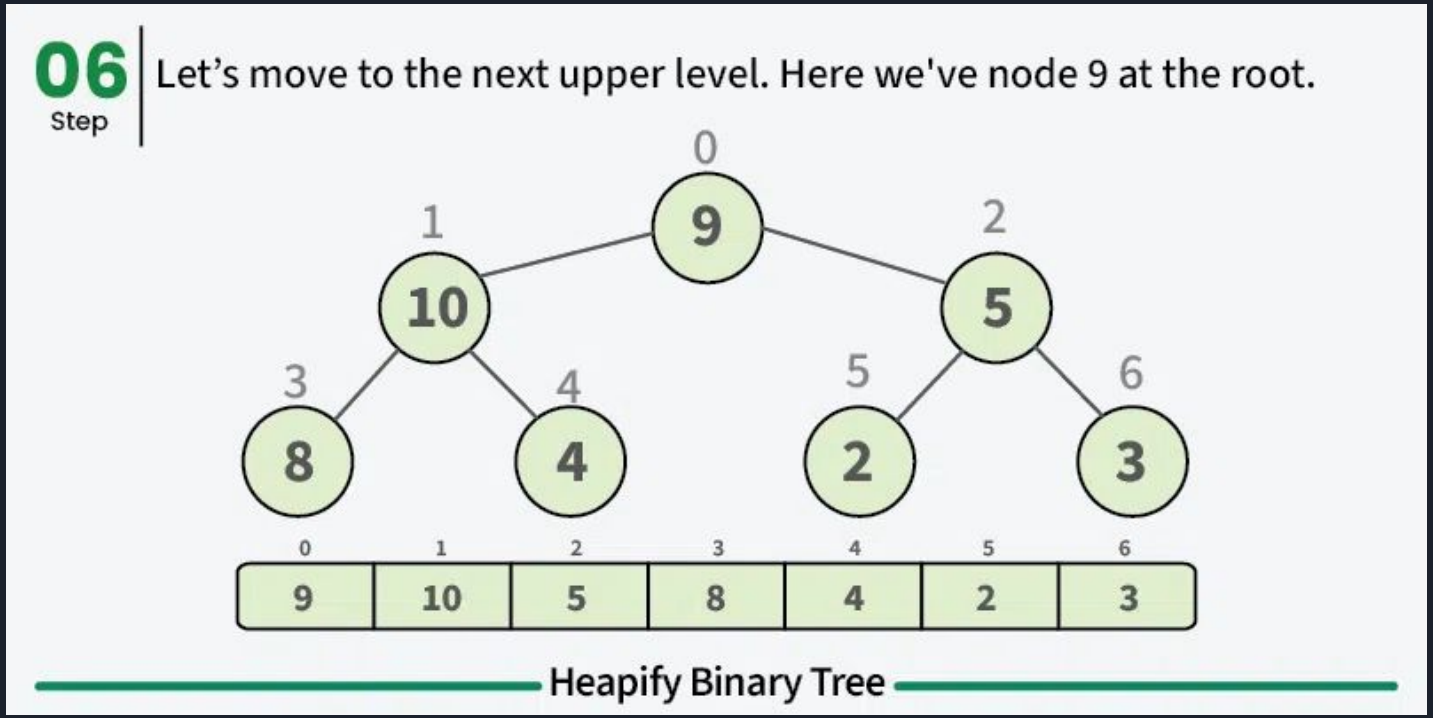
Algoritmos de Ordenamiento

- Heap Sort



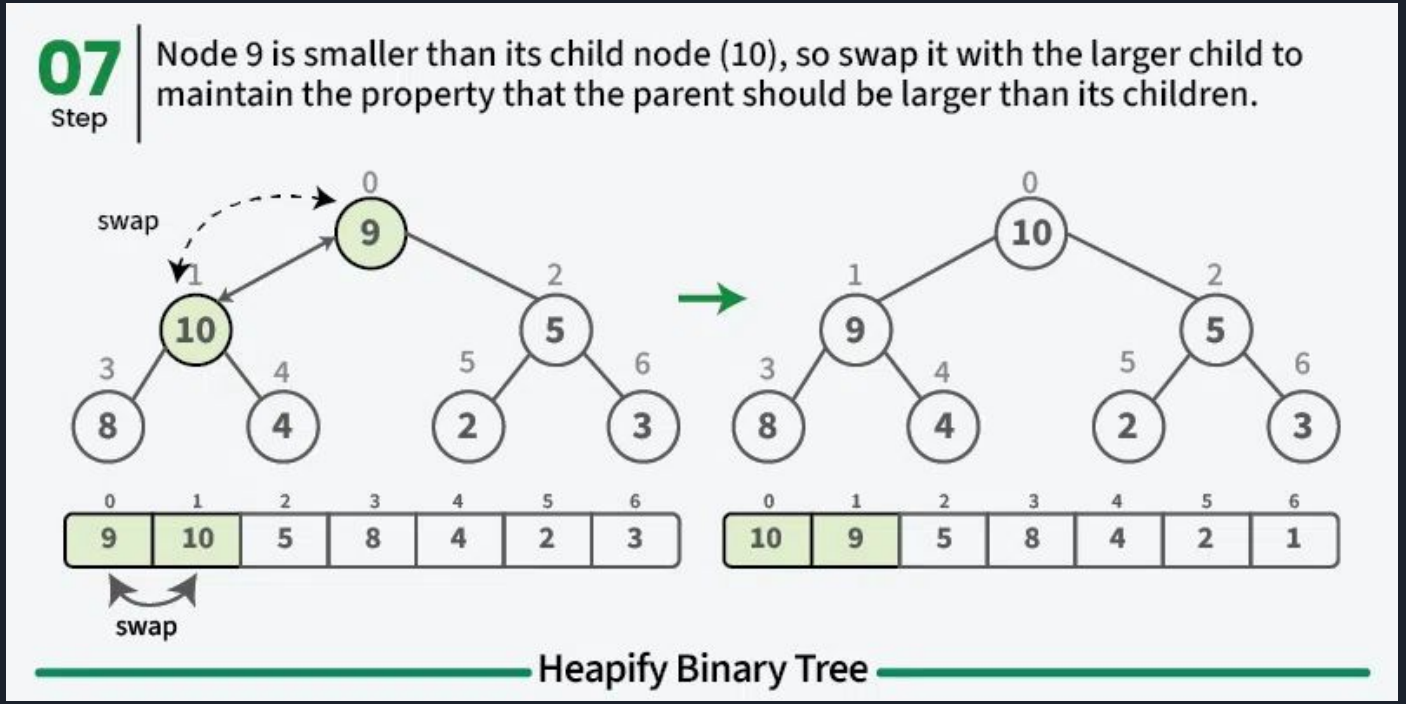
Algoritmos de Ordenamiento

- Heap Sort



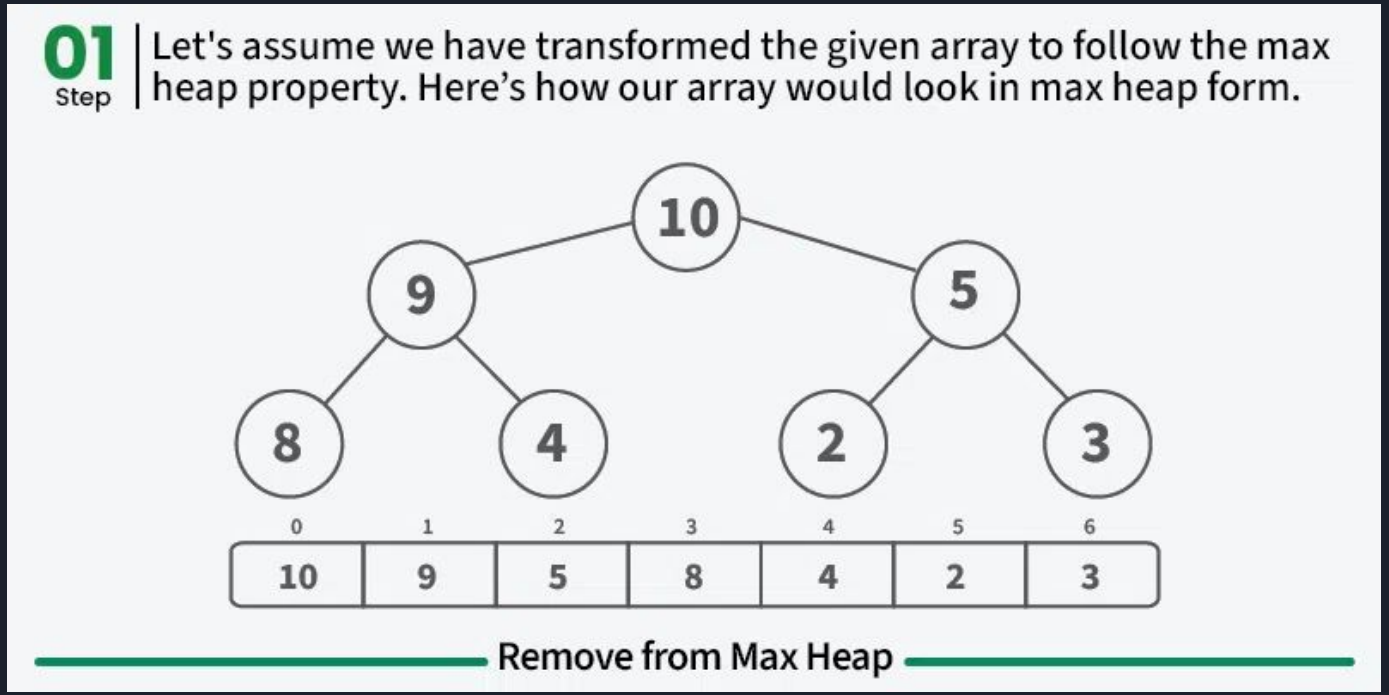
Algoritmos de Ordenamiento

- Heap Sort



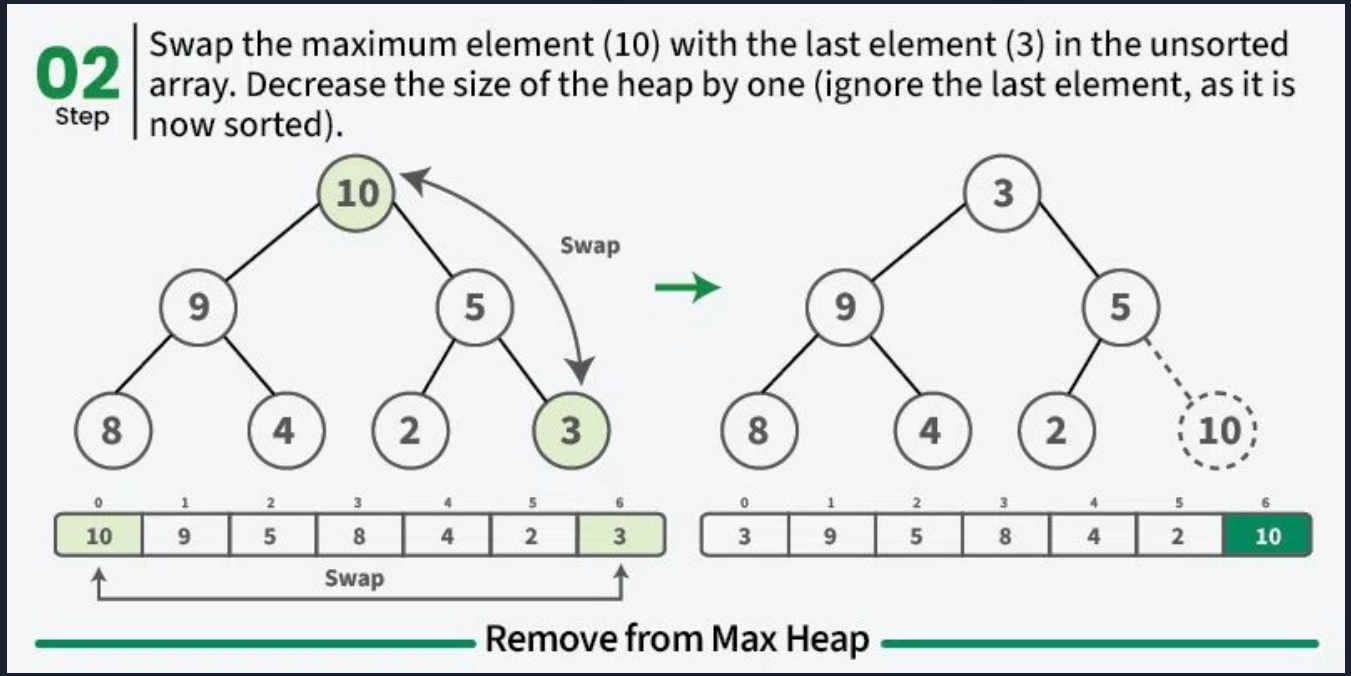
Algoritmos de Ordenamiento

- Heap Sort



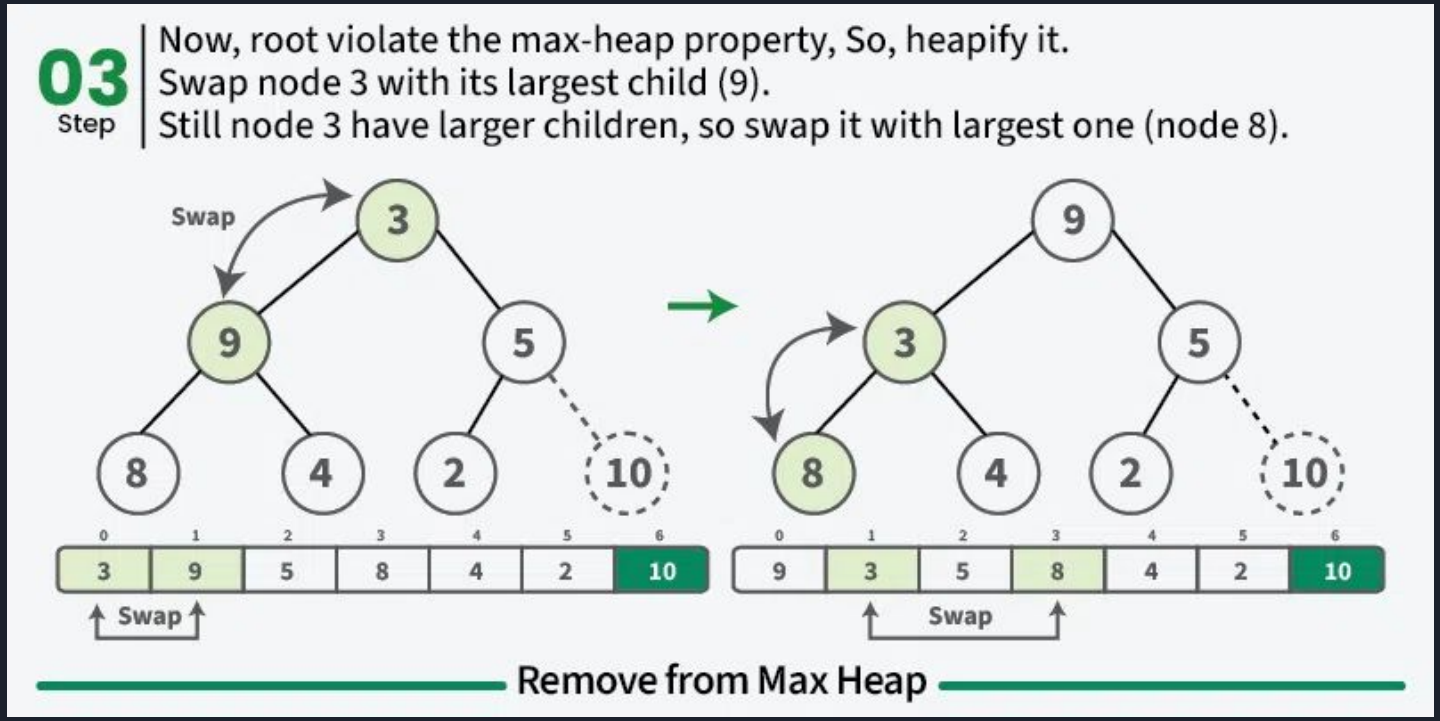
Algoritmos de Ordenamiento

- Heap Sort



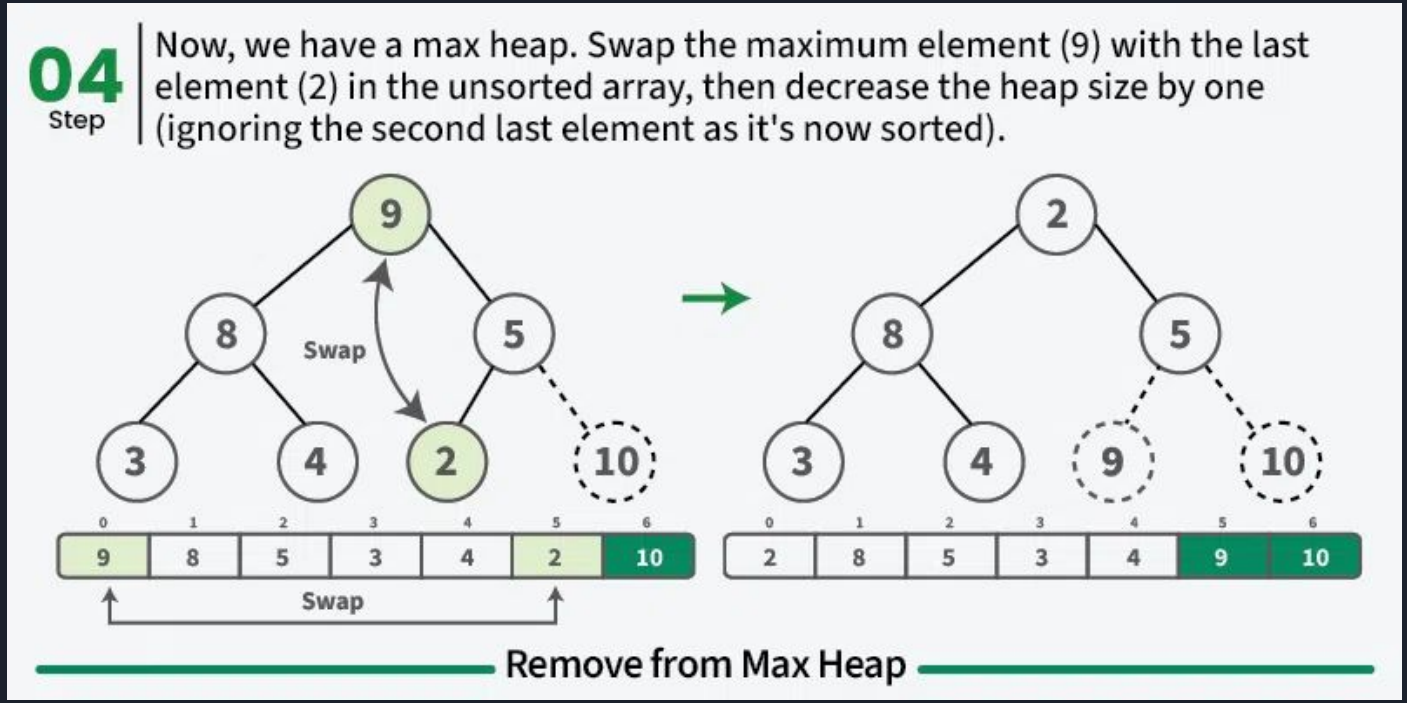
Algoritmos de Ordenamiento

- Heap Sort



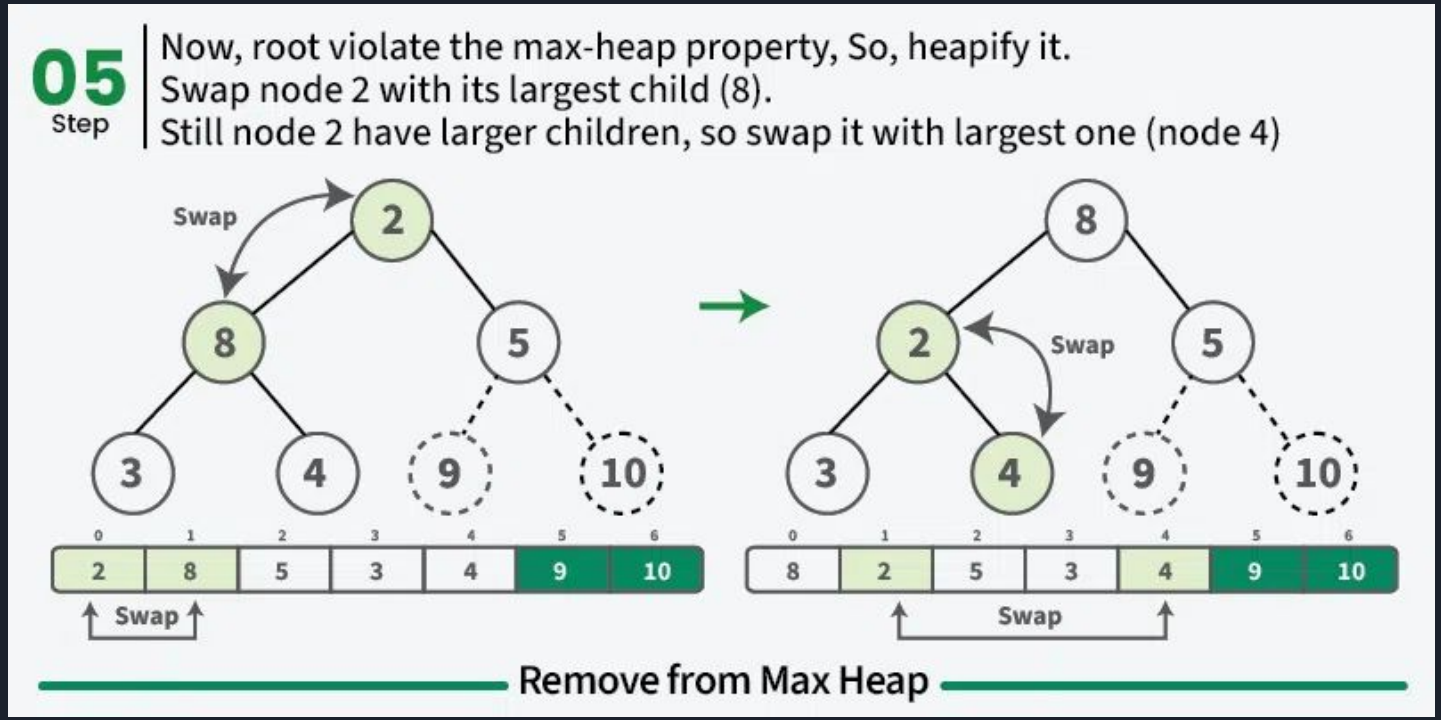
Algoritmos de Ordenamiento

- Heap Sort



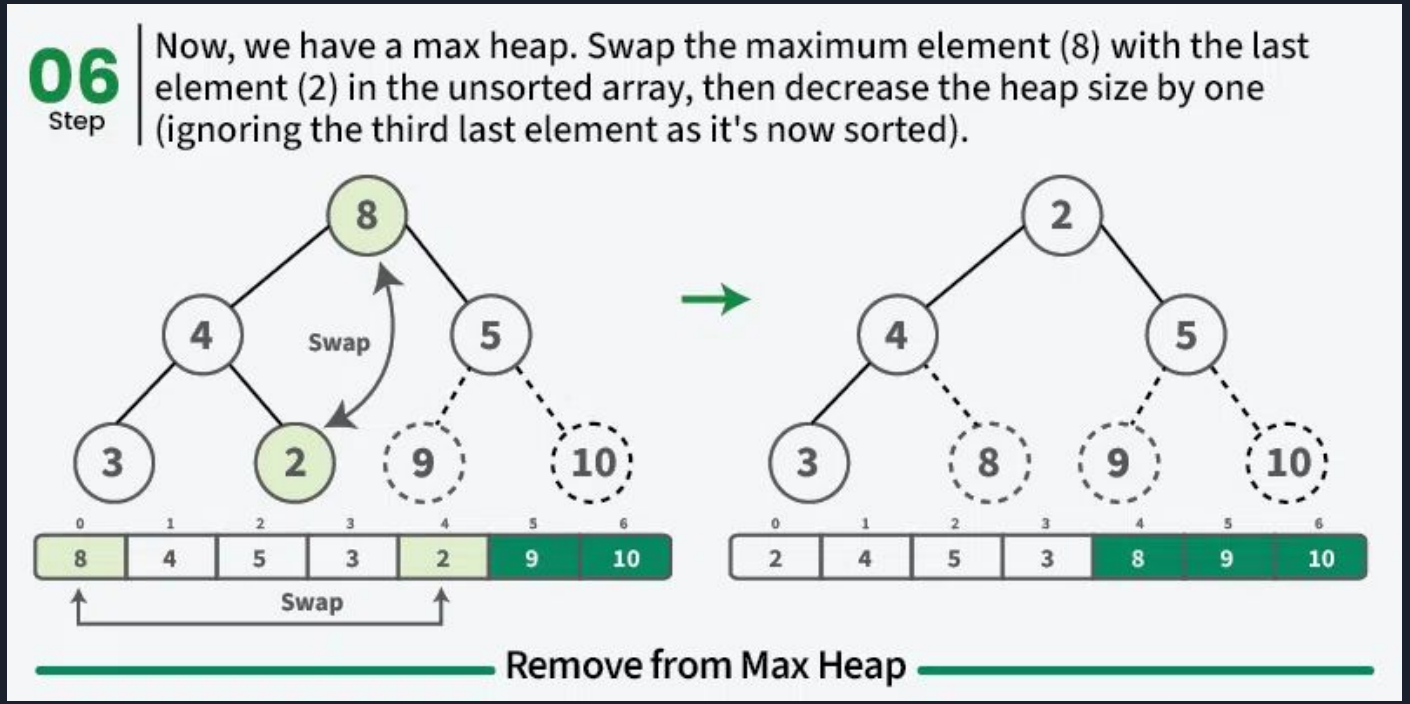
Algoritmos de Ordenamiento

- Heap Sort



Algoritmos de Ordenamiento

- Heap Sort



Algoritmos de Ordenamiento

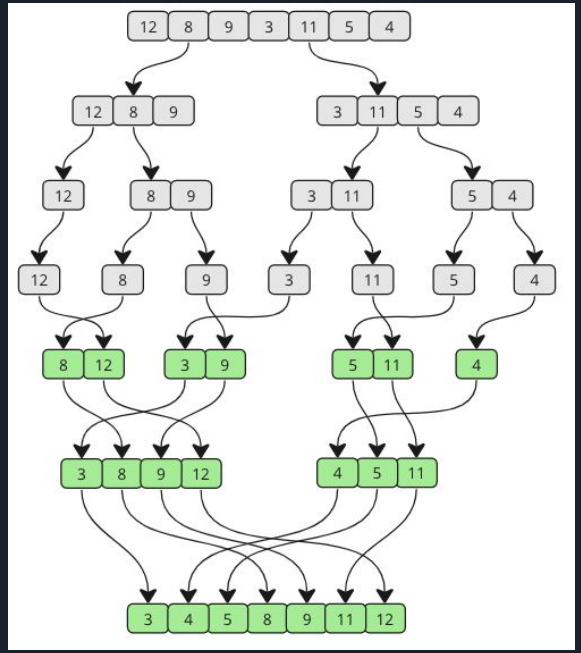
```cpp
void heapify(vector<int>& arr, int n, int i){

    // Initialize largest as root
    int largest = i;

    // left index = 2*i + 1
    int l = 2 * i + 1;

    // right index = 2*i + 2
    int r = 2 * i + 2;

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        swap(arr[i], arr[largest]);

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}
```

- Heap Sort → O(nlogn)

```cpp
void heapSort(vector<int>& arr){
    int n = arr.size();

    // Build heap (rearrange vector)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);

    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--) {

        // Move current root to end
        swap(arr[0], arr[i]);

        // Call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

Algoritmos de Ordenamiento

- Merge sort



Algoritmos de Ordenamiento

- Merge sort → O(n log n)

```
void mergeSort(vector<int>& arr, int left, int right)
{
    if (left >= right)
        return;

    int mid = left + (right - left) / 2;
    mergeSort(arr, left, mid);
    mergeSort(arr, mid + 1, right);
    merge(arr, left, mid, right);
}
```

```
void merge(vector<int>& arr, int left,
                    int mid, int right)
{
    int n1 = mid - left + 1;
    int n2 = right - mid;

    // Create temp vectors
    vector<int> L(n1), R(n2);

    // Copy data to temp vectors L[] and R[]
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];

    int i = 0, j = 0;
    int k = left;

    // Merge the temp vectors back
    // into arr[left..right]
    while (i < n1 && j < n2) {
        if (L[i] <= R[j]) {
            arr[k] = L[i];
            i++;
        }
        else {
            arr[k] = R[j];
            j++;
        }
        k++;
    }

    // Copy the remaining elements of L[],
    // if there are any
    while (i < n1) {
        arr[k] = L[i];
        i++;
        k++;
    }

    // Copy the remaining elements of R[],
    // if there are any
    while (j < n2) {
        arr[k] = R[j];
        j++;
        k++;
    }
}
```

Algoritmos de Ordenamiento

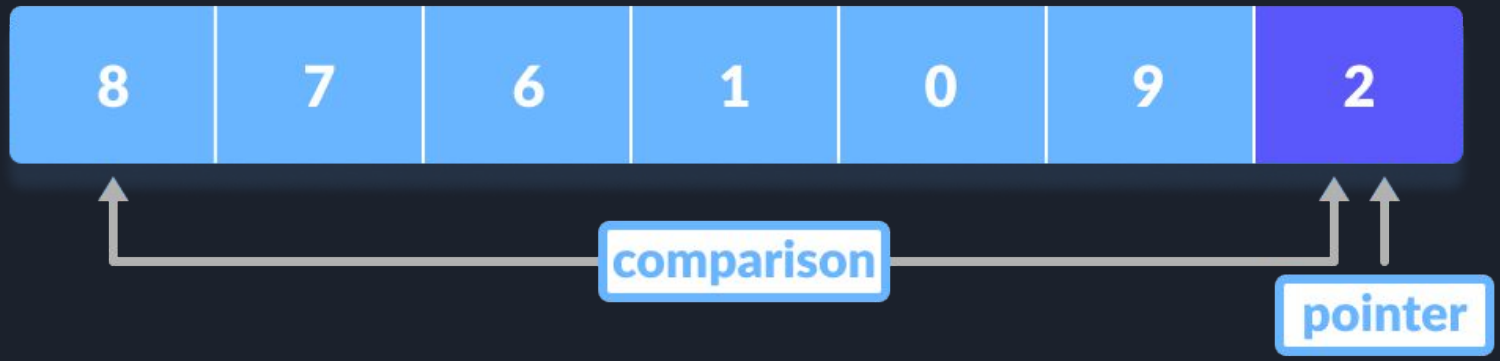
- Quick Sort

Debemos primero elegir un pivote

Algoritmos de Ordenamiento

- Quick Sort

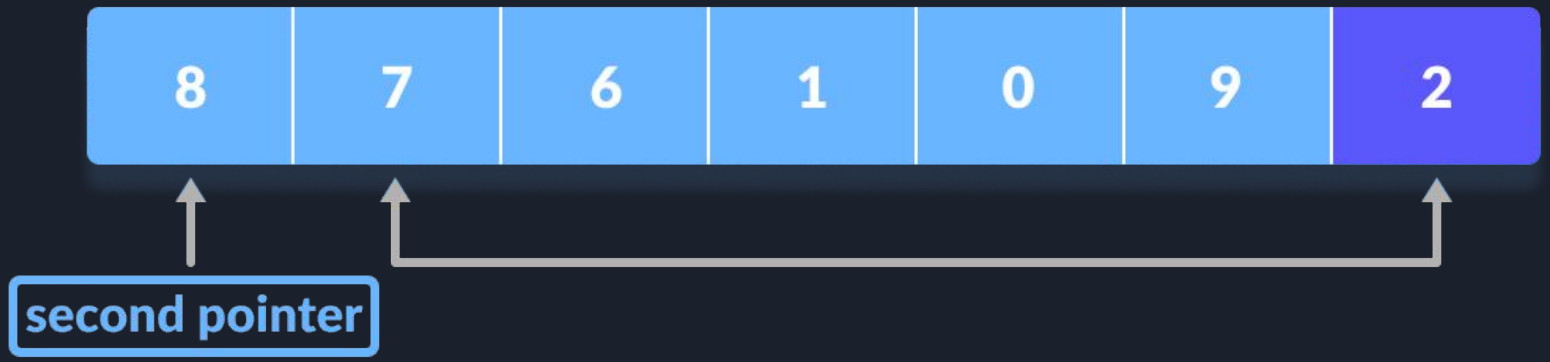
Un puntero está fijado en el elemento pivote. El elemento pivote se compara con los elementos comenzando desde el primer índice.



Algoritmos de Ordenamiento

- Quick Sort

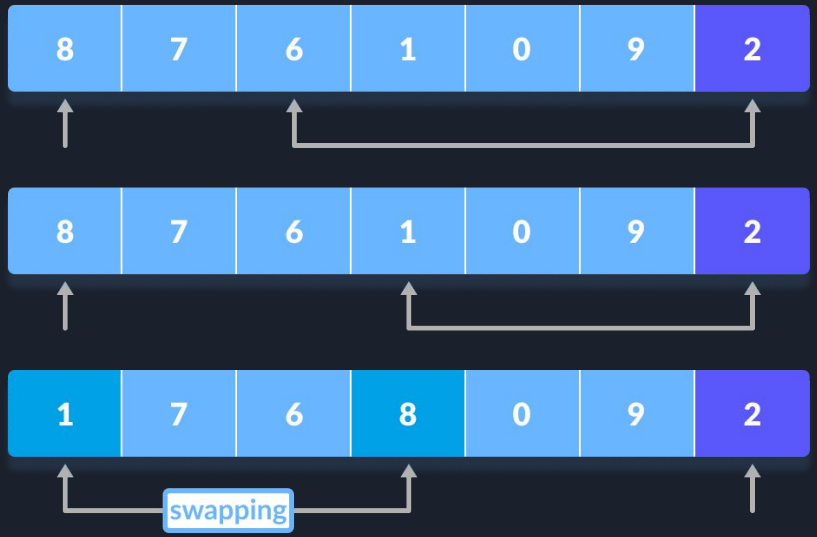
Si hay un elemento mayor al pivote, aparece entonces un segundo puntero y se fija en dicho elemento.



Algoritmos de Ordenamiento

- Quick Sort

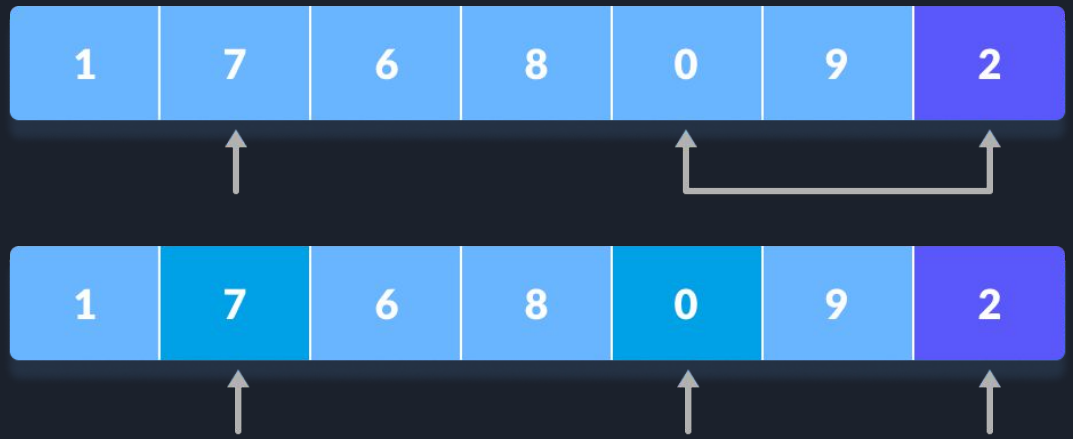
El pivote se compara con los otros elementos. Si se encuentra un elemento más pequeño que el pivote, ese elemento más pequeño se intercambia con el elemento mayor encontrado anteriormente.



Algoritmos de Ordenamiento

- Quick Sort

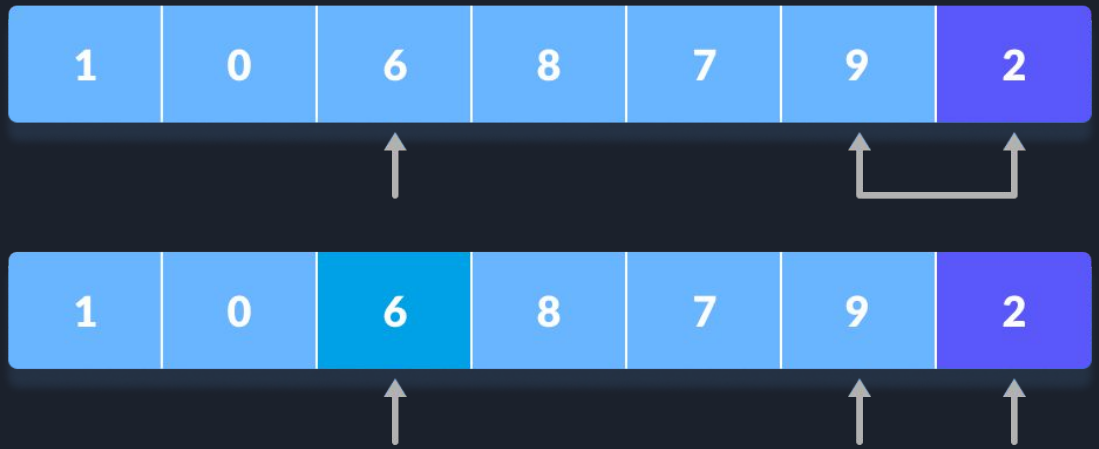
Se repite el proceso para establecer el siguiente elemento mayor como el segundo puntero. Luego, se intercambia con otro elemento más pequeño.



Algoritmos de Ordenamiento

- Quick Sort

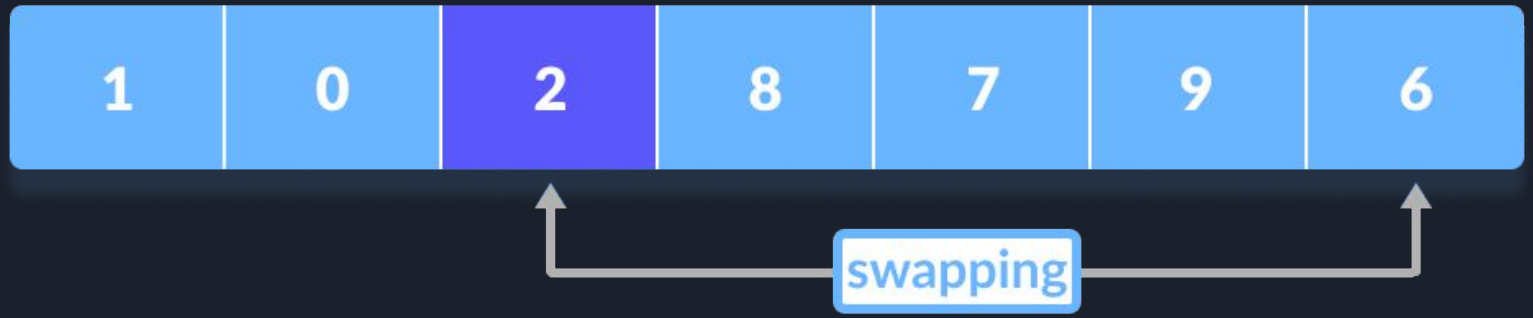
El proceso continúa hasta que se alcanza el penúltimo elemento.



Algoritmos de Ordenamiento

- Quick Sort

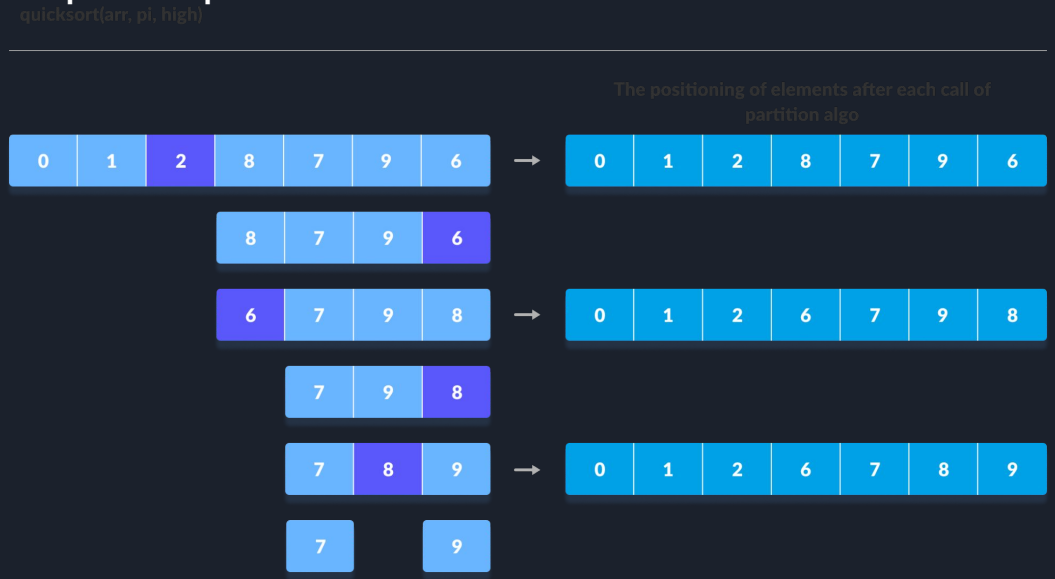
Finalmente, el elemento pivote se intercambia con el segundo puntero.



Algoritmos de Ordenamiento

- Quick Sort

Se eligen nuevamente elementos pivote para las subpartes izquierda y derecha por separado. Luego, se repite el paso 2.



Algoritmos de Ordenamiento

- Quick Sort (n^2 para este caso)

```
int partition(vector<int>& arr, int low, int high) {

    // Choose the pivot
    int pivot = arr[high];

    // Index of smaller element and indicates
    // the right position of pivot found so far
    int i = low - 1;

    // Traverse arr[;ow..high] and move all smaller
    // elements on left side. Elements from low to
    // i are smaller after every iteration
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }

    // Move pivot after smaller elements and
    // return its position
    swap(arr[i + 1], arr[high]);
    return i + 1;
}
```

```
void quickSort(vector<int>& arr, int low, int high) {

    if (low < high) {

        // pi is the partition return index of pivot
        int pi = partition(arr, low, high);

        // Recursion calls for smaller elements
        // and greater or equals elements
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Algoritmos de Ordenamiento

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(1)
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$	O(1)
Heap Sort	O(n log(n))	O(n log(n))	O(n log(n))	O(1)
Quick Sort	O(n log(n))	O(n log(n))	$O(n^2)$	O(n)
Merge Sort	O(n log(n))	O(n log(n))	O(n log(n))	O(n)
Bucket Sort	O(n +k)	O(n +k)	$O(n^2)$	O(n)
Radix Sort	O(nk)	O(nk)	O(nk)	O(n + k)
Count Sort	O(n +k)	O(n +k)	O(n +k)	O(k)
Shell Sort	O(n log(n))	O(n log(n))	$O(n^2)$	O(1)
Tim Sort	O(n)	O(n log(n))	O(n log (n))	O(n)
Tree Sort	O(n log(n))	O(n log(n))	$O(n^2)$	O(n)
Cube Sort	O(n)	O(n log(n))	O(n log(n))	O(n)

Clases de algoritmos

- Divide and conquer: Se divide el problema en sub-problemas más pequeños y luego dichas soluciones se organizan para formar la solución completa al problema. → ejemplo: quick sort

- Greedy: Son algoritmos que miran aquellas alternativas inmediatas y de mejor progreso. No evalúan sobre la secuencia completa del algoritmo. → ejemplo: Prim

Clases de algoritmos

- Programación dinámica: Algoritmos que utilizan parte ya calculada de un subproblema para resolver el problema completo. Utilizan estructuras de datos de almacenamiento para llevar recuento de las soluciones intermedias y no recalcularlas. → Revisar fibonacci como ejemplo “de partida” para recordar cómo funciona!