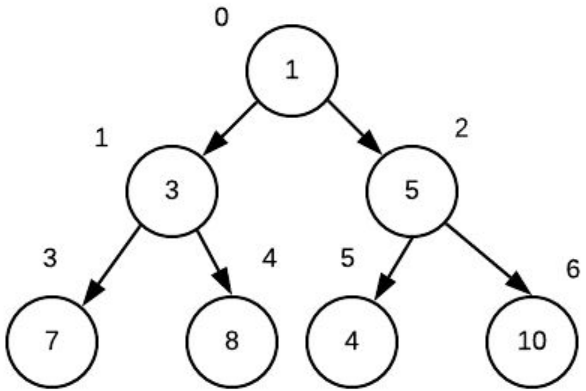
Heap Sort

Curso: Algoritmos y Estructura de Datos
Profesor: Luis Talavera

Integrantes:
- Diego Enciso
- Nelson Soberon

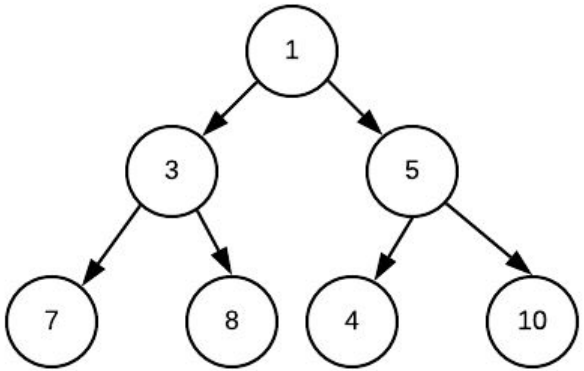
Ejemplo

Arr = [1, 3, 5, 7, 8, 4, 10]



Ejemplo

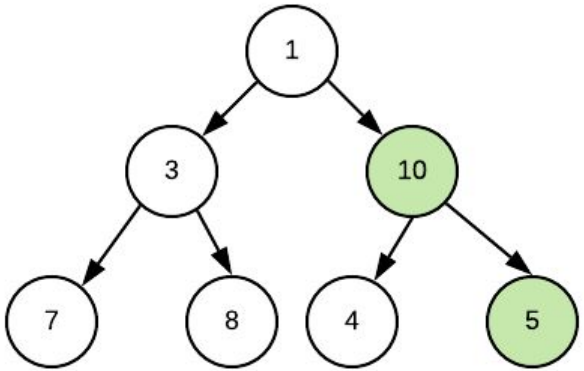
Arr = [1, 3, 5, 7, 8, 4, 10]



Heapify!

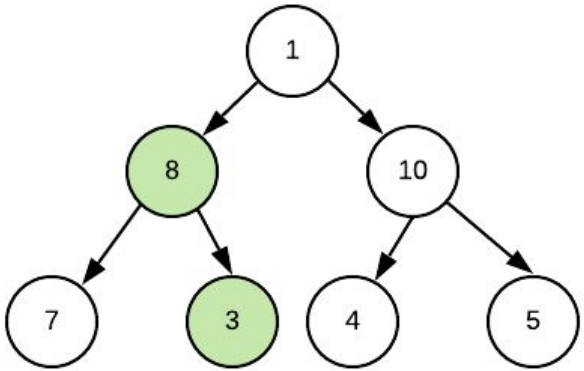
Ejemplo

Arr = [1, 3, 10, 7, 8, 4, 5]



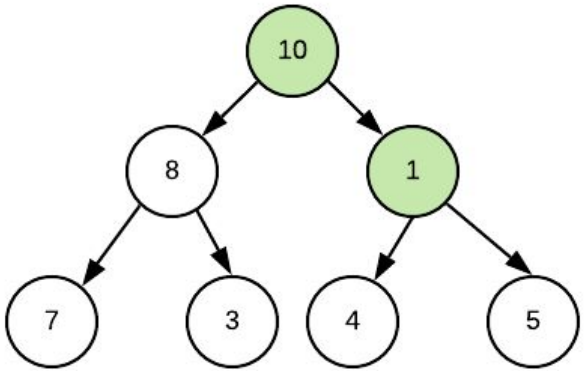
Ejemplo

Arr = [1, 8, 10, 7, 3, 4, 5]



Ejemplo

Arr = [10, 8, 1, 7, 3, 4, 5]

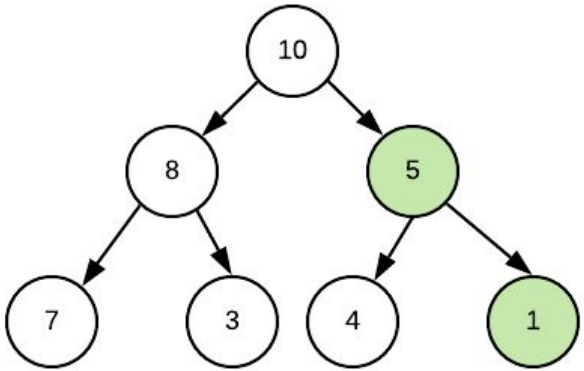


Ejemplo

Arr = [10, 8, 5, 7, 3, 4, 1]

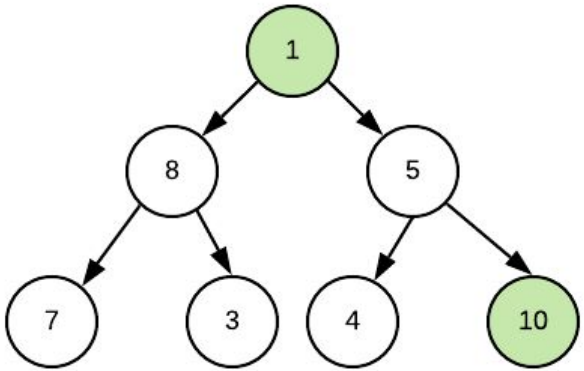
Max Heap

Completado!



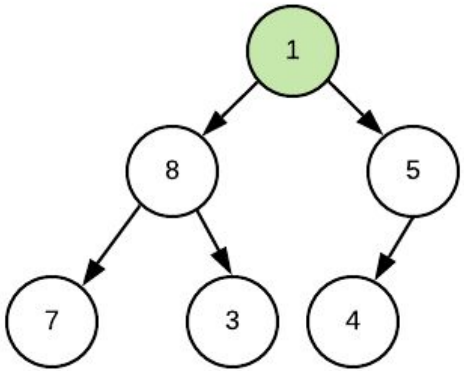
Ejemplo

Arr = [1, 8, 5, 7, 3, 4, 10]



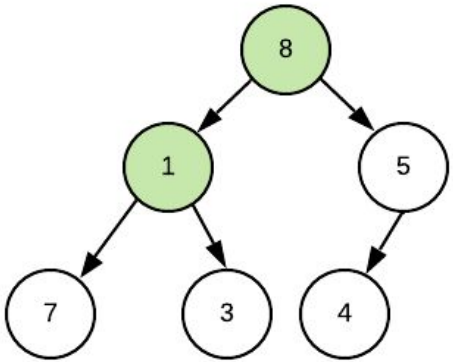
Ejemplo

Arr = [1, 8, 5, 7, 3, 4, 10]



Ejemplo

Arr = [8, 1, 5, 7, 3, 4, 10]

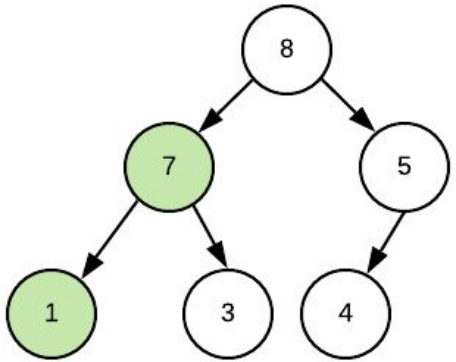


Ejemplo

Arr = [8, 7, 5, 1, 3, 4, 10]

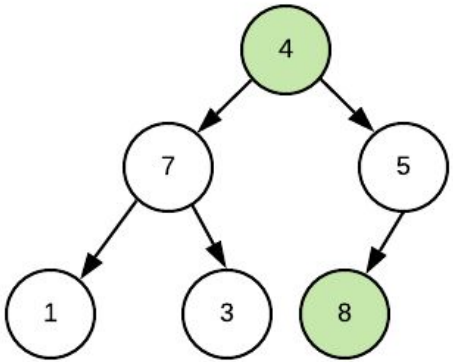
Max Heap

Completado!



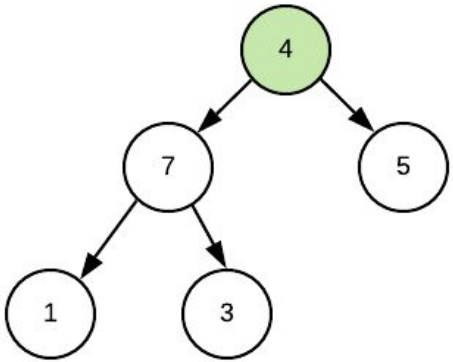
Ejemplo

Arr = [4, 7, 5, 1, 3, 8, 10]



Ejemplo

Arr = [4, 7, 5, 1, 3, 8, 10]

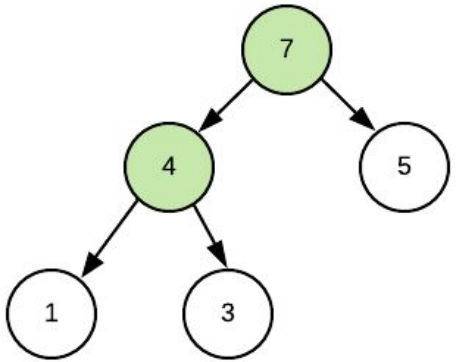


Ejemplo

Arr = [7, 4, 5, 1, 3, 8, 10]

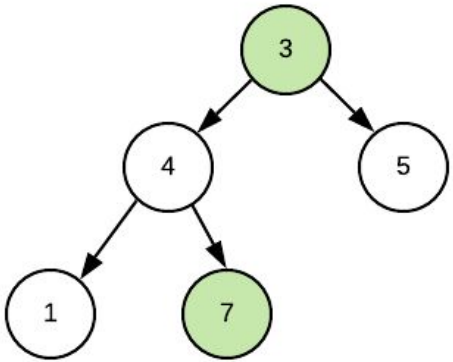
Max Heap

Completado!



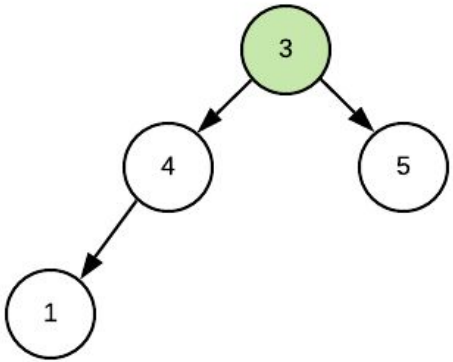
Ejemplo

Arr = [3, 4, 5, 1, 7, 8, 10]



Ejemplo

Arr = [3, 4, 5, 1, 7, 8, 10]

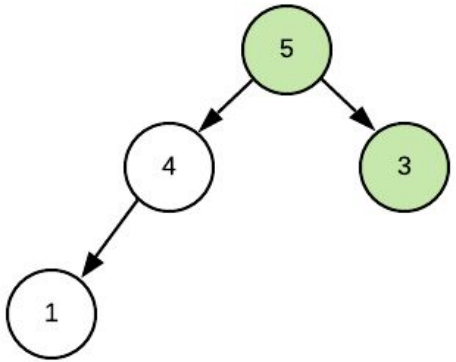


Ejemplo

Arr = [5, 4, 3, 1, 7, 8, 10]

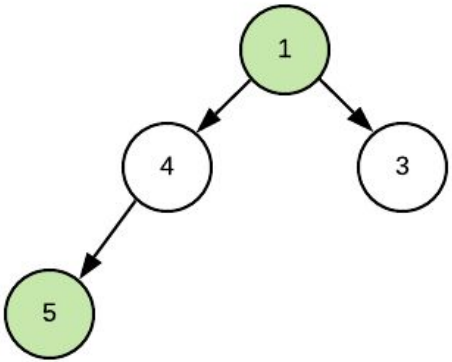
Max Heap

Completado



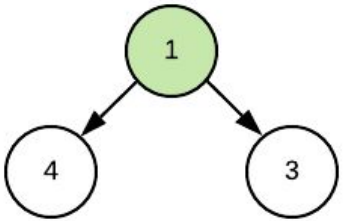
Ejemplo

Arr = [1, 4, 3, 5, 7, 8, 10]



Ejemplo

Arr = [1, 4, 3, 5, 7, 8, 10]

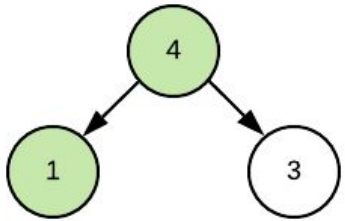


Ejemplo

Arr = [4, 1, 3, 5, 7, 8, 10]

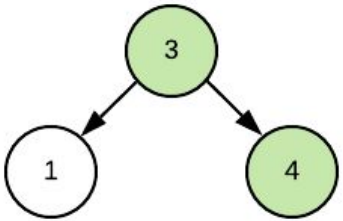
Max Heap

Completado!



Ejemplo

Arr = [3, 1, 4, 5, 7, 8, 10]

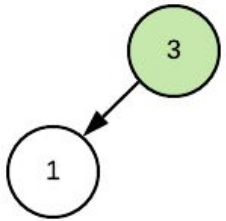


Ejemplo

Arr = [3, 1, 4, 5, 7, 8, 10]

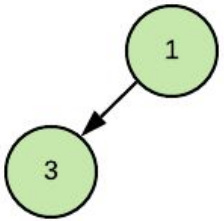
Max Heap

Completado



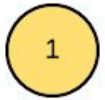
Ejemplo

Arr = [1, 3, 4, 5, 7, 8, 10]



Ejemplo

Arr = [1, 3, 4, 5, 7, 8, 10]



- Complejidad (tiempo): O(nlog(n))
- Complejidad (espacio): O(1)
- Inestable

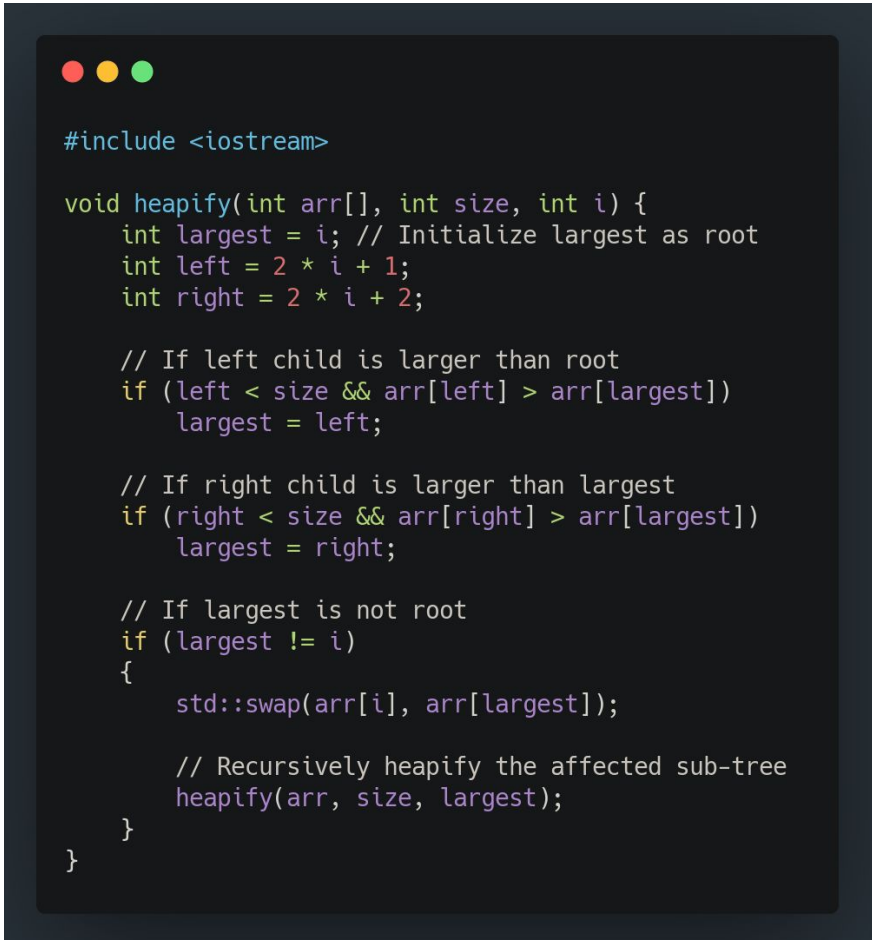
¿Cuándo debe usarse? y ¿qué limitaciones tiene?

Cuando usarse:

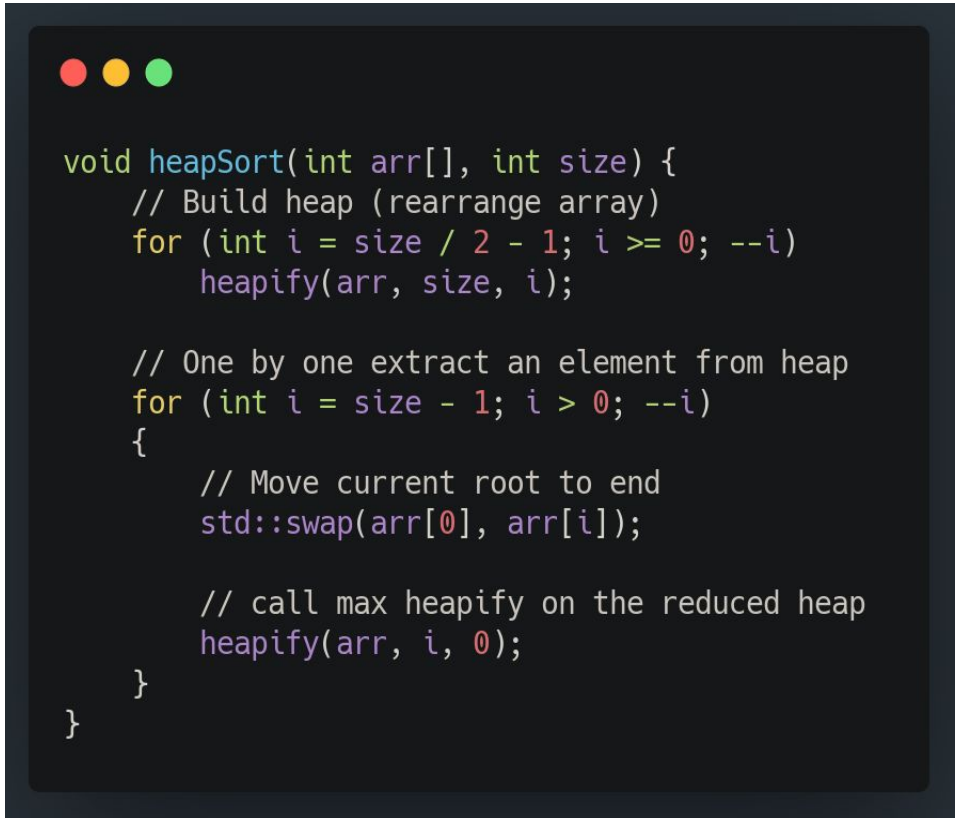
- Cuando se quiere extraer el mínimo y el máximo.
- Cuando se tiene un sistema embebido en el que el espacio es limitado.

Limitaciones:

- Es inestable.
- Tiene factores constantes que lo vuelven más lento que el Quicksort o Mergesort por ejemplo.



```cpp
#include <iostream>

void heapify(int arr[], int size, int i) {
    int largest = i; // Initialize largest as root
    int left = 2 * i + 1;
    int right = 2 * i + 2;

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

    // If right child is larger than largest
    if (right < size && arr[right] > arr[largest])
        largest = right;

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

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



```
void heapSort(int arr[], int size) {
    // Build heap (rearrange array)
    for (int i = size / 2 - 1; i >= 0; --i)
        heapify(arr, size, i);

    // One by one extract an element from heap
    for (int i = size - 1; i > 0; --i)
    {
        // Move current root to end
        std::swap(arr[0], arr[i]);

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

```cpp
void printArray(int arr[], int size) {
    for (int i = 0; i < size; ++i)
        std::cout << arr[i] << " ";
    std::cout << "\n";
}

int main() {
    int arr[] = {1, 3, 5, 7, 8, 4, 10};
    int size = sizeof(arr) / sizeof(arr[0]);

    heapSort(arr, size);

    std::cout << "Sorted array: ";
    printArray(arr, size);
}
```

Resultado

```
diego@archlinux ~/Documents/
Original array: 1 3 5 7 8 4 10
Sorted array  : 1 3 4 5 7 8 10
```

Preguntas

1. ¿Cómo se podría hacer un HeapSort de forma descendiente?
2. ¿Si todos los elementos del array son iguales, cuál sería la complejidad de tiempo de HeapSort?

GRACIAS!