Algoritmo de Kruskal

Algorithm 1: Kruskal

input : $G = (V, E)$ grafo, $\omega : E \to \mathbb{R}$.
output: T árbol generador de peso mínimo de G.
ordenar E como $e_1, \ldots, e_m$ tal que $\omega(e_1) \leq \omega(e_2) \leq \ldots \leq \omega(e_m)$;
/* Inicializar T */
$T \leftarrow (V, \emptyset)$;
for $i \leftarrow 1$ **to** m **do**
    if *los extremos de* e_i *están en componentes distintas de* T **then**
        $T \leftarrow T + e_i$
return(T)

Algoritmo de Prim

Algorithm 2: Prim

input : $G = (V, E)$ grafo, $\omega : E \to \mathbb{R}$, $r \in V$.

output: T árbol generador de peso mínimo de G.

/* Inicializar T */

$T \leftarrow (\{r\}, \emptyset)$;

while T *no es árbol generador* **do**

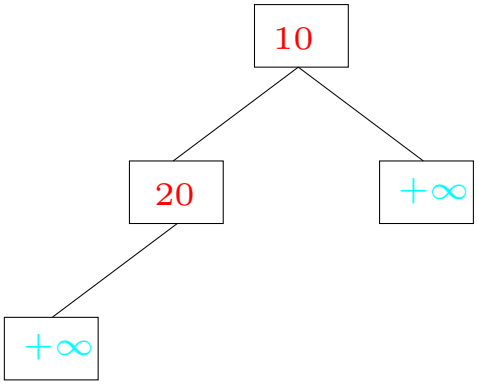
  $uv \leftarrow \operatorname{argmin}\{\omega(\tilde{u}\tilde{v}) : \tilde{u} \in V(T), \tilde{v} \notin V(T)\}$;

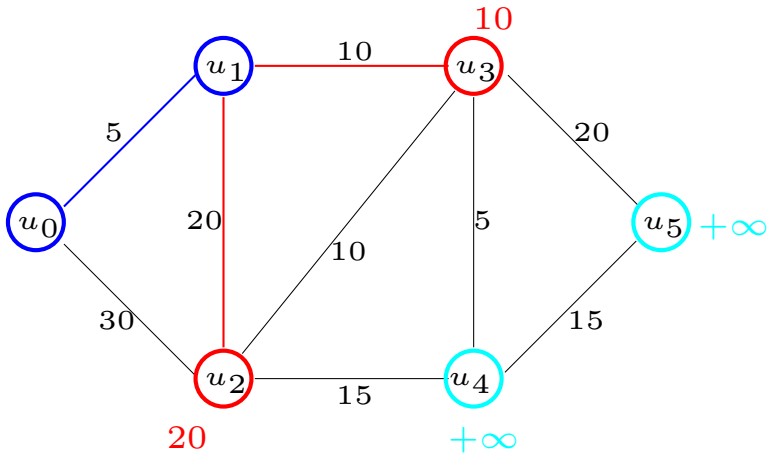
  $V(T) \leftarrow V(T) \cup \{v\}$;

  $E(T) \leftarrow E(T) \cup \{uv\}$;

return(T)

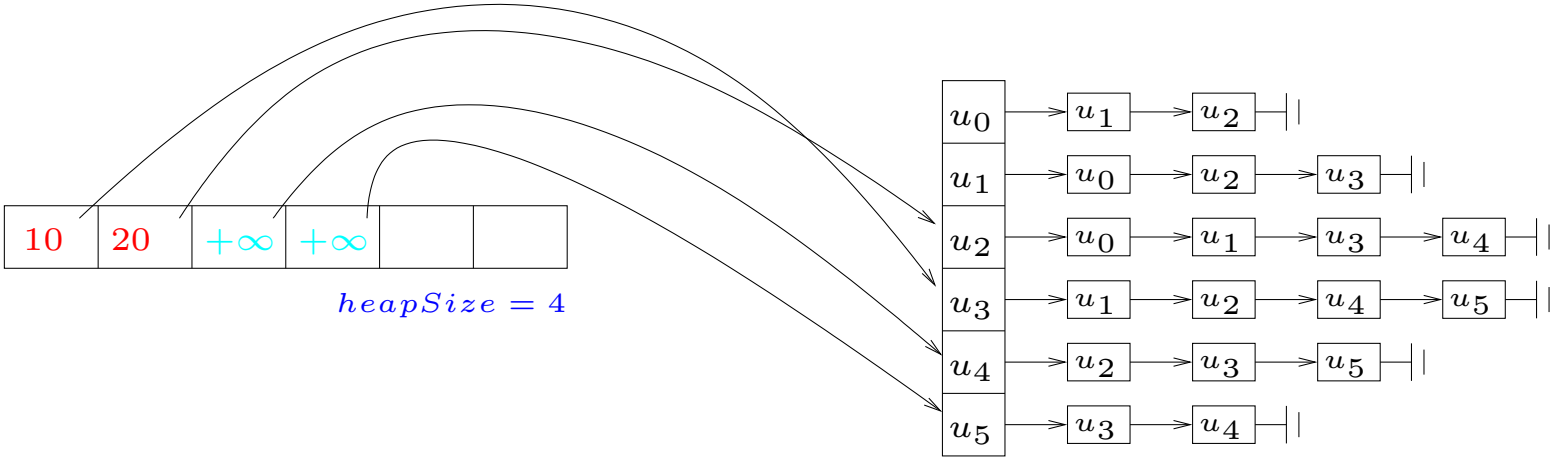
Implementación del Algoritmo de Prim





Heap

Grafo



Algorithm 3: Prim

input : Representación de $G = (V, E)$ grafo, $\omega : E \to \mathbb{R}$, $r \in V$.
output: Representación de T árbol generador de peso mínimo de G.

```
// Inicializar Heap
for u ∈ V do
    Q[u].key ← +∞;   Q[u].node ← u;   π[u] ← Nil;
Q[r].key ← 0;
HeapBuild(Q);
/* heapSize: variable global igual al número de items en el heap Q.   */
heapSize ← |V|;
while la heap Q no esta vacía do
    u ← HeapExtractMin(Q);
    for v ∈ V tal que uv ∈ E do
        if v ∈ Q y ω(uv) < Q[v].key then
            π[v] ← u;
            HeapDecreaseKey(Q, v, ω(uv));
return(π);
```

Heaps

Algorithm 4: HeapExtractMin

input : Q heap.

output: Extrae item del heap Q con key de valor mínimo.

```
/* heapSize: variable global igual al número de items en el heap Q.  */
if heapSize < 1 then
    return("Error: Underflow");
min ← Q[1].node;
Q[1] ← Q[heapSize];
heapSize ← heapSize − 1;
Heapify(Q,1);
return(min)
```

Algorithm 5: Heapify

input : Q heap, i índice de un item del heap.

output: Actualiza el heap Q de manera que el "sub-árbol" enraizado en el item i satisfaga la "propiedad heap".

```
/* heapSize: variable global igual al número de items en el heap Q.  */
```

$l \leftarrow 2i$; $r \leftarrow 2i + 1$;

if $l \leq heapSize$ y $Q[l].key < Q[i].key$ **then**

  $smallest \leftarrow l$;

else

  $smallest \leftarrow i$;

if $r \leq heapSize$ y $Q[r].key < Q[smallest].key$ **then**

  $smallest \leftarrow r$;

if $smallest \neq i$ **then**

  Intercambiar $Q[i] \leftrightarrow Q[smallest]$;

  `Heapify`$(Q, smallest)$;

Algorithm 6: HeapDecreaseKey

input : Q heap, i índice de un item del heap, key nueva llave.
output: Decrementa a key el valor de la llave del item i del heap Q.

if $key > Q[i].key$ **then**
  return *"Error: nueva llave es mayor que llave actual."*;
$Q[i].key \leftarrow key$;
while $i > 1$ *y* $Q[\lfloor i/2 \rfloor].key > Q[i].key$ **do**
  Intercambiar $Q[i] \leftrightarrow Q[\lfloor i/2 \rfloor]$;
  $i \leftarrow \lfloor i/2 \rfloor$;