Colas de prioridades

IIC2133

1-2016

El tipo de datos

Un computador puede ejecutar varias aplicaciones al mismo tiempo:

- asigna una prioridad (clave) a los eventos asociados con las aplicaciones
- … y siempre elige para procesar a continuación el evento con la prioridad (clave) más alta

Un tipo de datos apropiado —la **cola de prioridades** — tiene dos operaciones:

- sacar el máximo
- insertar

Usos

Simulaciones

Programación de tareas

Computación numérica

Algoritmo de ordenación

Algoritmos de búsqueda en grafos

Algoritmo de compresión de datos

API
(para MaxPQ)

```
maxPQ()

maxPQ(int max)

maxPQ(Key[] a)

void insert(Key v)

Key max()

Key delMax()

boolean isEmpty()

int size()
```

Cuatro implementaciones básicas

Arreglo o **lista ligada**

… y ya sea **ordenado** o **no ordenado**:

- … colas de prioridades pequeñas
- … cuando una de las dos operaciones predomina
- … o cuando podemos hacer algunas suposiciones acreca del orden de las claves

O() del tiempo de ejecución en el peor caso

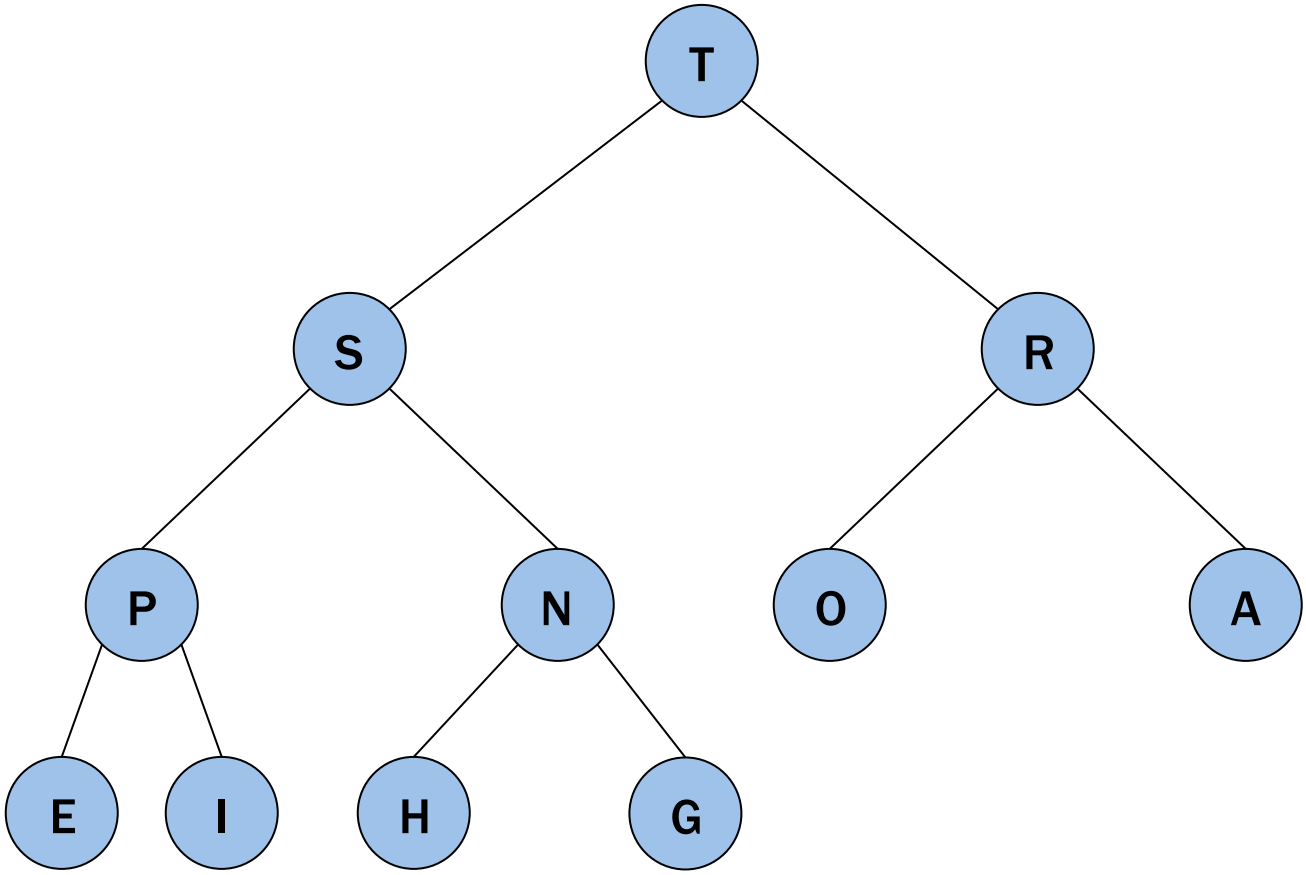
estructura	**`insert`**	**`delMax`**
arreglo ordenado	*N*	1
arreglo no ordenado	1	*N*
heap	**log*N***	**log*N***
imposible	1	1

operación	cola no ordenada	cola ordenada
insert P	P	P
insert Q	P Q	P Q
insert E	P Q E	E P Q
delMax → Q	P E	E P
insert X	P E X	E P X
insert A	P E X A	A E P X
insert M	P E X A M	A E M P X
delMax → X	P E M A	A E M P
insert P	P E M A P	A E M P P
insert L	P E M A P L	A E L M P P
insert E	P E M A P L E	A E E L M P P
delMax → P	E E M A P L	A E E L M P

Heaps
(binarios)

Un árbol binario es un ***heap*** si la clave en cada nodo es mayor que las claves en los dos hijos de ese nodo —**propiedad de *heap***

La clave más grande en un árbol binario *heap* está en la raíz —se demuestra por inducción sobre el tamaño del árbol



Representaciones

Representación mediante **punteros**:

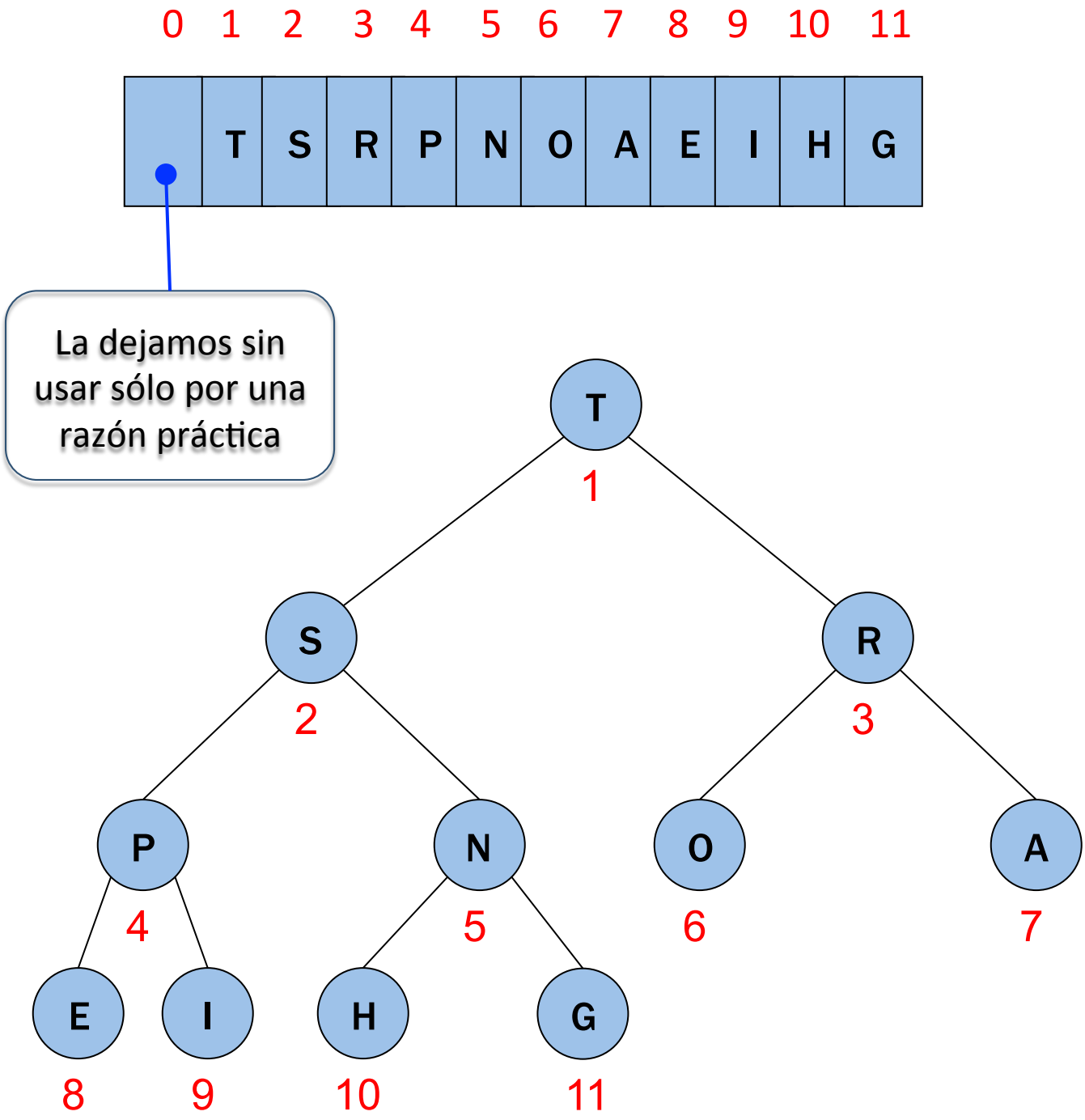
- tres punteros asociados a cada clave, apuntando uno al padre y uno a cada uno de los hijos

Conviene usar un árbol binario *completo*, como el de la fig. anterior, que permite

… representación mediante **arreglo**:

- colocamos los nodos en orden por nivel,
- … con la raíz en la posición 1,
- … sus hijos en las posiciones 2 y 3,
- … los hijos de éstos en las posiciones 4, 5, 6 y 7, etc.

Un ***heap*** (**binario**) es una colección de claves dispuestas en un árbol binario *heap* completo, representado en orden por nivel en un arreglo



Propiedades

El padre de un nodo en la posición k está en la posición $\lfloor k/2 \rfloor$

Los dos hijos de un nodo en la posición k están en las posiciones $2k$ y $2k+1$

En vez de usar punteros explícitos,

… podemos subir y bajar por el árbol haciendo operaciones aritméticas simples sobre los índices del arreglo

La altura de un árbol binario completo de tamaño N es $\lfloor \log N \rfloor$

Cómo funcionan las operaciones de *heap*

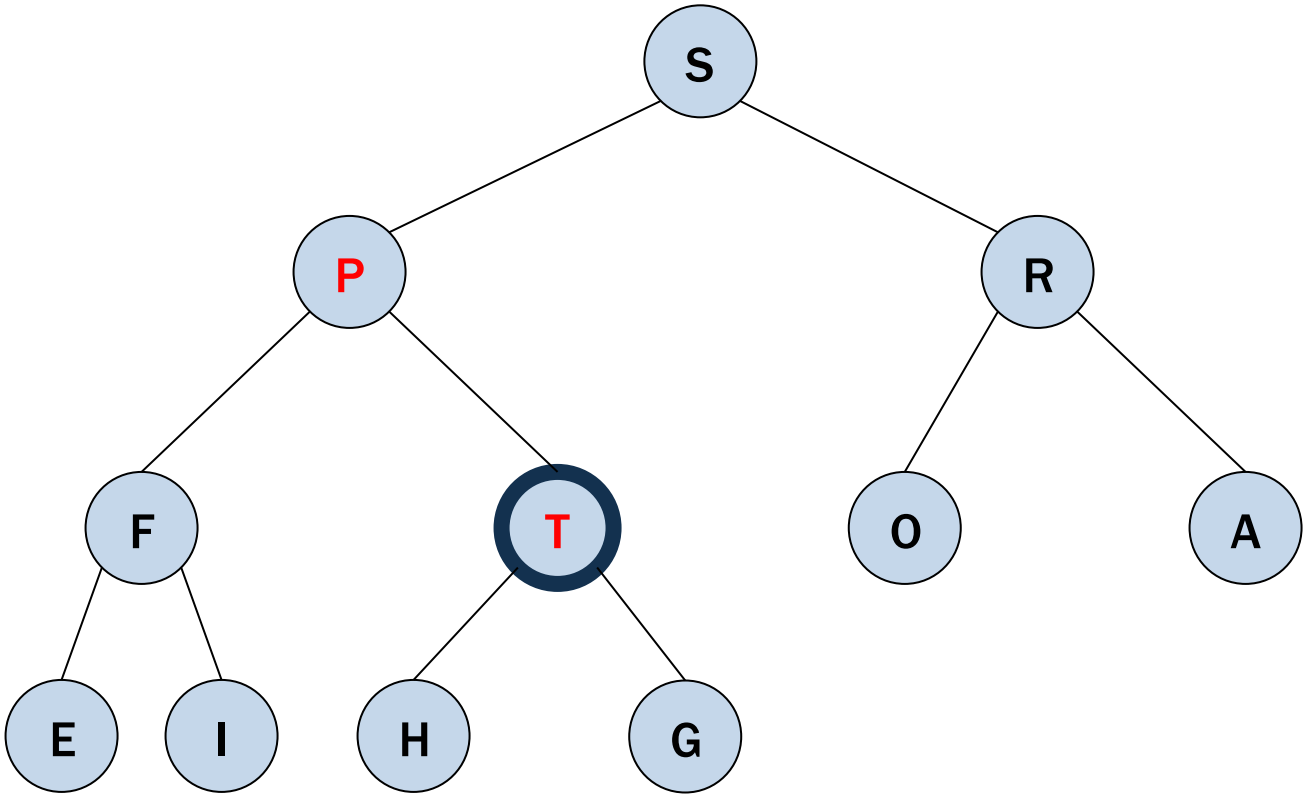
Primero, hacen un cambio simple que podría violar la condición (propiedad) de *heap*:

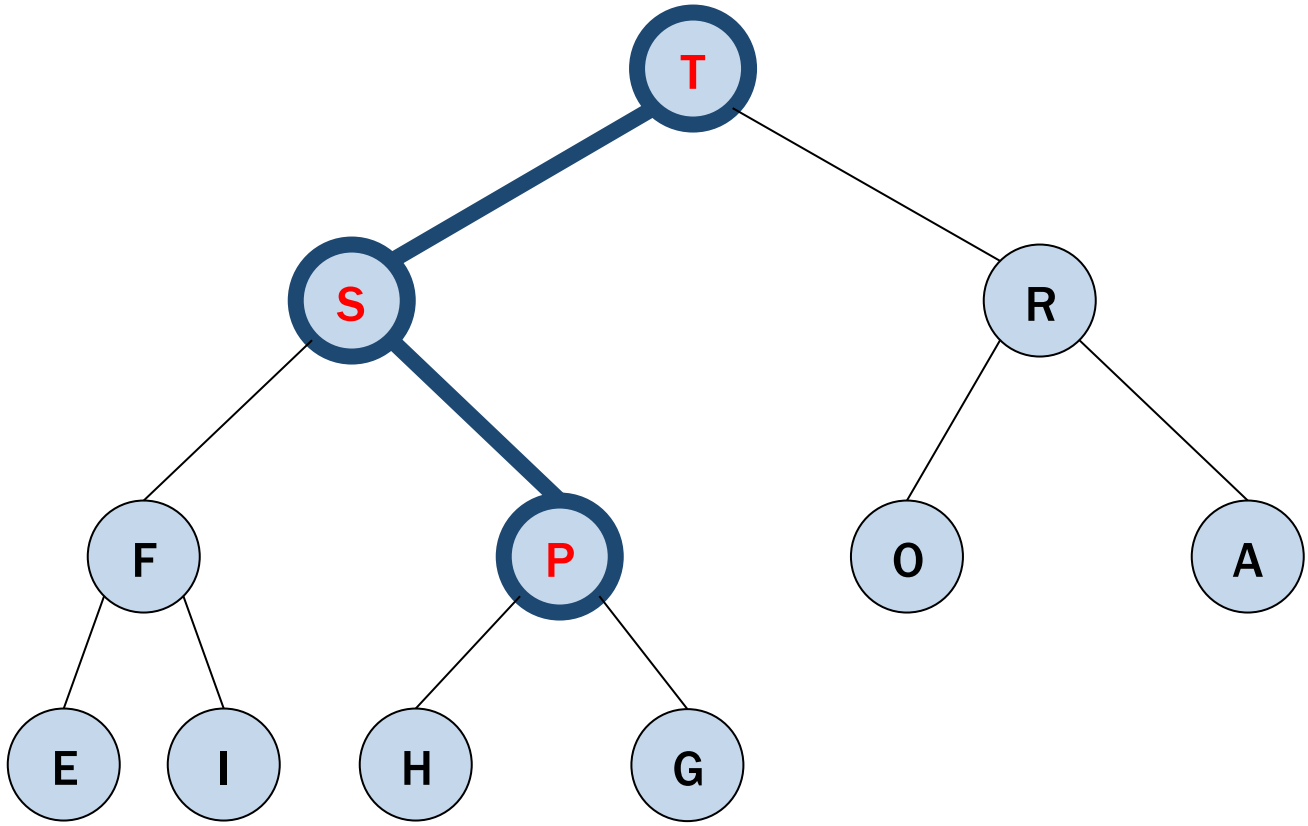
- p.ej., en la próxima diapositiva, el nodo con clave *T*, está violando la propiedad de heap c/r a su padre

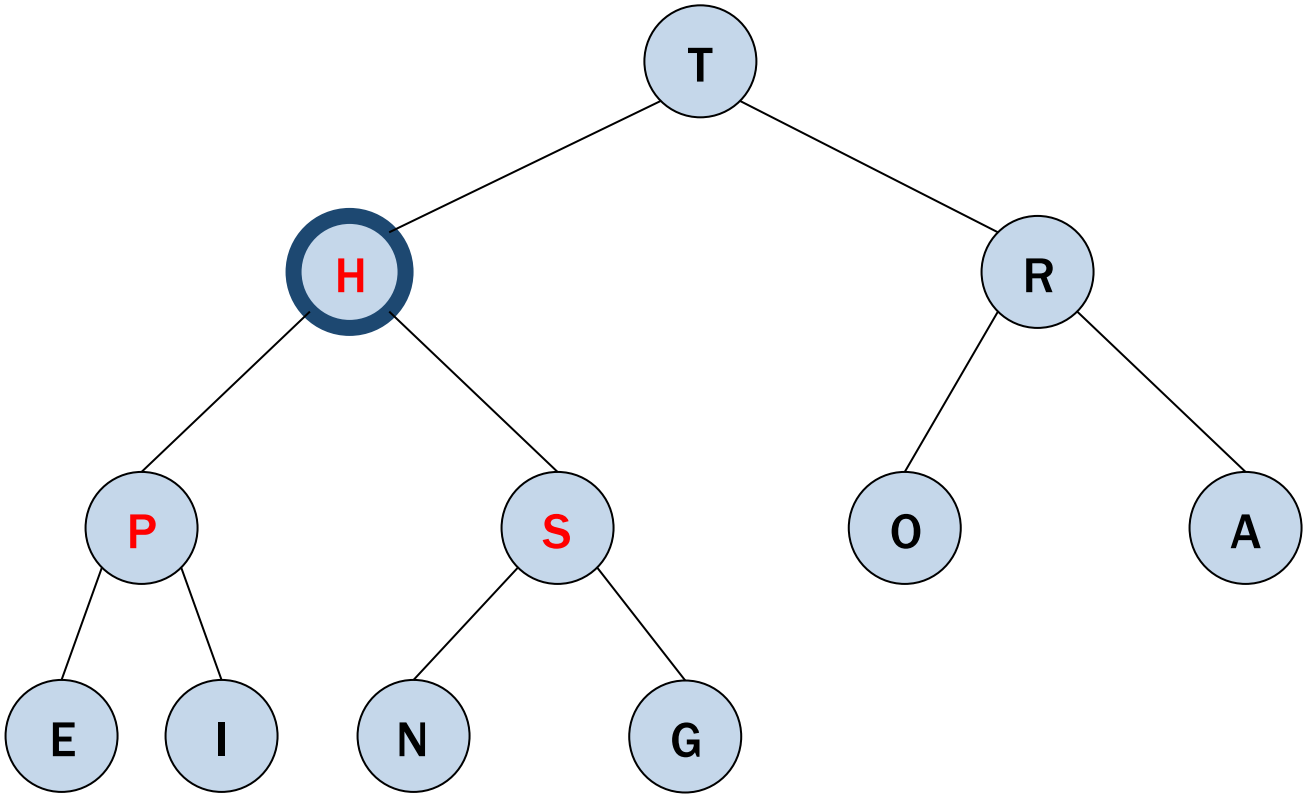
  … y en la subsiguiente, el nodo con clave *H*, la está violando c/r a sus hijos

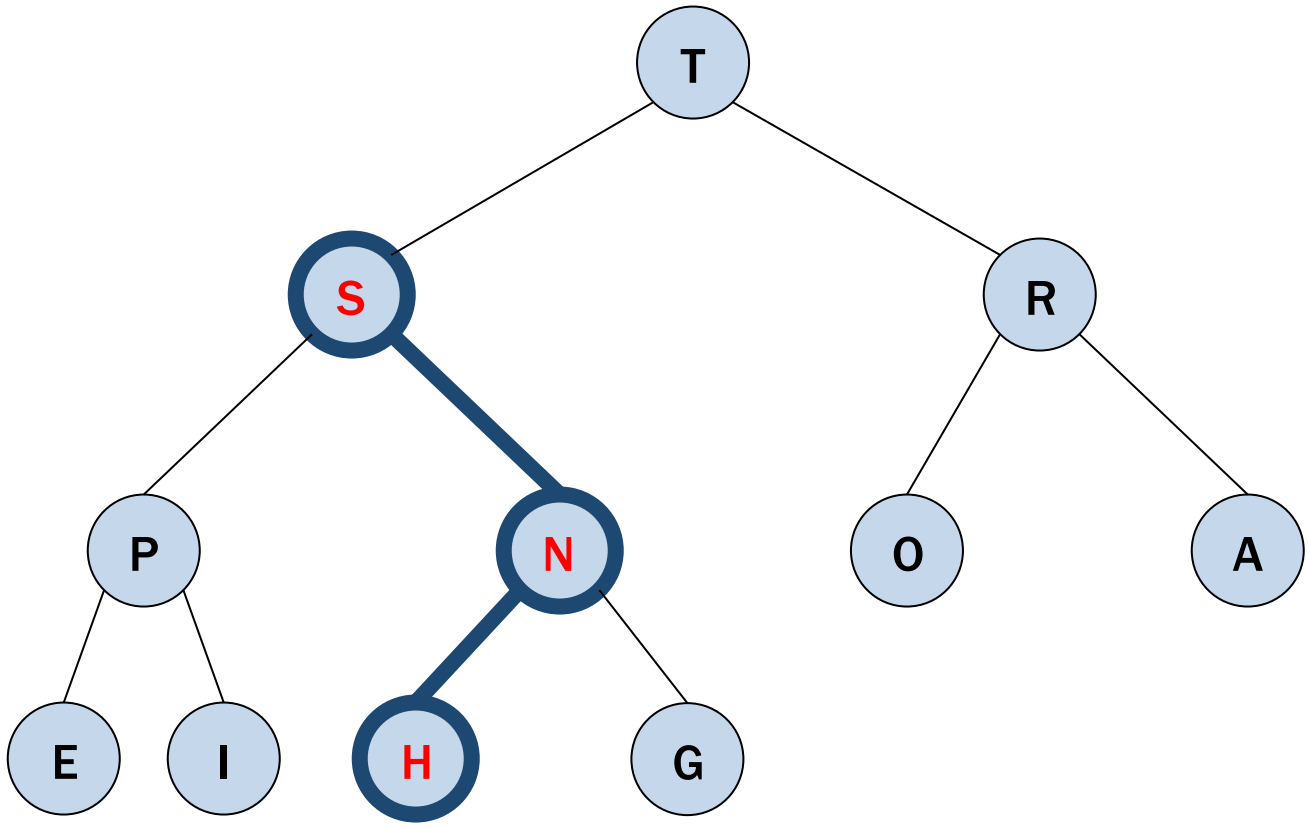
… luego, se mueven por el *heap* cambiándolo como sea necesario para asegurar que la condición de *heap* se cumpla en todo el *heap* —**restauración de la propiedad de *heap***:

- el nodo con clave *T* "sube" por el árbol —intercambiándose con su padre— hasta quedar correctamente ubicado
- el nodo con clave *H* "baja" por el árbol, intercambiándose cada vez con el mayor de sus hijos


S
P
R
F
T
O
A
E
I
H
G


T
S
R
F
P
O
A
E
I
H
G


T
H
R
P
S
O
A
E
I
N
G


T
S
R
P
N
O
A
E
I
H
G

Restauración de la propiedad de *heap*

Si la prioridad de un nodo aumenta o un nodo nuevo es agregado al final del heap, hay que subir por el heap

```
void up(int k):
   while (k > 1 && less(k/2, k))
      exch(k/2, k)
      k = k/2
```

Si la prioridad de un nodo disminuye, bajamos por el heap

```
void down(int k):
   while (2*k ≤ N)
      j = 2*k
      if (j < N && less(j, j+1)) j = j+1
      if (!less(k, j)) break
      exch(k, j)
      k = j
```

Las operaciones *insert* y *remove the maximum*

```
void insert(Key v):
      Q[++N] = v
      up(N)

Key delMax():
      Key max = Q[1]
      exch(1, N--)
      Q[N+1] = null
      down(1)
      return max
```

En una cola de prioridades de *N* claves, los algoritmos de *heap* requieren

… no más de 1 + log*N* comparaciones para *insert*

… y no más de 2log*N* para *remove the maximum*

Esta mejora significativa frente a las implementaciones básicas puede hacer la diferencia entre resolver un problema y no ser capaz ni siquiera de abordarlo