Colas de prioridad y heaps

Flavia Bonomo - Algoritmos II

1er. cuatrimestre 2015

Colas de prioridad

- Numerosas aplicaciones: sistemas operativos, algoritmos de scheduling, etc.
- La prioridad en general la expresamos con un entero, pero puede ser cualquier otro tipo α con un orden $<_\alpha$ asociado.
- Correspondencia entre máxima prioridad y un valor máximo o mínimo del valor del tipo α.

Colas de prioridad

Colas de prioridad

Colas de prioridad

Operaciones que nos interesan:

- Obtener el elemento de máxima prioridad.
- Insertar un elemento con una prioridad.
- Eliminar el elemento de máxima prioridad.
- *Aumentar/disminuir la prioridad de un elemento.*
- *Construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades.*

TAD Cola de prioridad($\alpha, <_\alpha$)

Observadores básicos

- vacía?: colaPrior($\alpha, <_\alpha$) $\rightarrow$ bool
- próximo: colaPrior($\alpha, <_\alpha$) c $\rightarrow \alpha$ $\quad$ ($\neg$vacía?(c))
- desencolar: colaPrior($\alpha, <_\alpha$) c $\rightarrow$ colaPrior($\alpha, <_\alpha$) $\quad$ ($\neg$vacía?(c))

Generadores

- vacía: $\rightarrow$ colaPrior($\alpha, <_\alpha$)
- encolar: $\alpha \times$ colaPrior($\alpha, <_\alpha$) $\rightarrow$ colaPrior($\alpha, <_\alpha$)

Otras Operaciones

- $\cdot =_{\text{colaPrior}(\alpha, <_\alpha)} \cdot$: colaPrior($\alpha, <_\alpha$)$\times$ colaPrior($\alpha, <_\alpha$) $\rightarrow$ bool

Posibles representaciones

Lista o arreglo desordenado

- Obtener el elemento de máxima prioridad.
- Insertar un elemento con una prioridad.
- Eliminar el elemento de máxima prioridad.
- *Aumentar/disminuir la prioridad de un elemento.*
- *Construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades.*

Posibles representaciones

Lista o arreglo desordenado

- Obtener el elemento de máxima prioridad. $O(n)$
- Insertar un elemento con una prioridad. $O(1)$
- Eliminar el elemento de máxima prioridad. $O(n)$
- *Aumentar/disminuir la prioridad de un elemento.* $O(1)$
- *Construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades.* $O(n)$

Posibles representaciones

Lista o arreglo ordenado por prioridad

- Obtener el elemento de máxima prioridad.
- Insertar un elemento con una prioridad.
- Eliminar el elemento de máxima prioridad.
- *Aumentar/disminuir la prioridad de un elemento.*
- *Construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades.*

Posibles representaciones

Lista o arreglo ordenado por prioridad

- Obtener el elemento de máxima prioridad. $O(1)$
- Insertar un elemento con una prioridad. $O(n)$
- Eliminar el elemento de máxima prioridad. $O(1)$
- *Aumentar/disminuir la prioridad de un elemento.* $O(n)$
- *Construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades.* $O(n\log(n))$

Posibles representaciones

Lista o arreglo ordenado por prioridad (caso cola/pila)

- Obtener el elemento de máxima prioridad. $O(1)$
- Insertar un elemento con una prioridad. $O(n)$ ($O(1)^*$)
- Eliminar el elemento de máxima prioridad. $O(1)$
- *Aumentar/disminuir la prioridad de un elemento.* $O(n)$
- *Construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades.* $O(n \log(n))$

* Ojo! Acá en el caso cola/pila hay un invariante muy fuerte que es que las inserciones vienen ordenadas por prioridad....

Posibles representaciones

Heap (o “montón”)

- Obtener el elemento de máxima prioridad.
- Insertar un elemento con una prioridad.
- Eliminar el elemento de máxima prioridad.
- *Aumentar/disminuir la prioridad de un elemento.*
- *Construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades.*

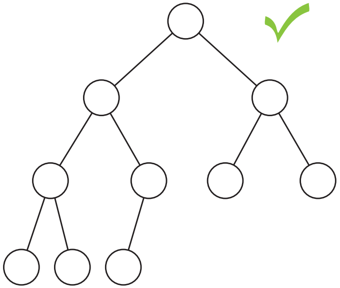
Posibles representaciones

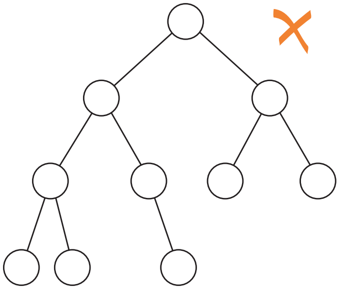
Heap (o "montón")

- Obtener el elemento de máxima prioridad. $O(1)$
- Insertar un elemento con una prioridad. $O(\log(n))$
- Eliminar el elemento de máxima prioridad. $O(\log(n))$
- *Aumentar/disminuir la prioridad de un elemento.* $O(\log(n))$
- *Construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades.* $O(n)$

Heaps

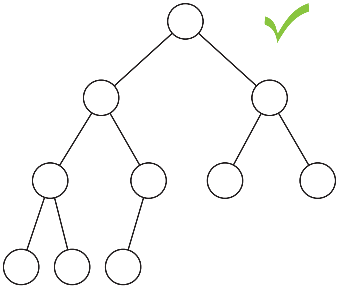
- Árbol binario completo salvo quizás el último nivel, que se llena de izquierda a derecha.

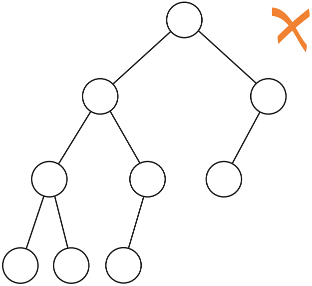




Heaps

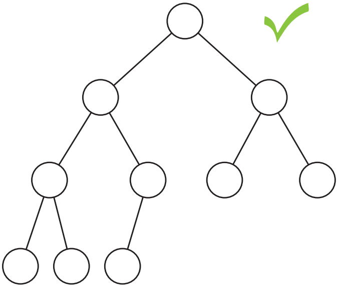
- Árbol binario completo salvo quizás el último nivel, que se llena de izquierda a derecha.

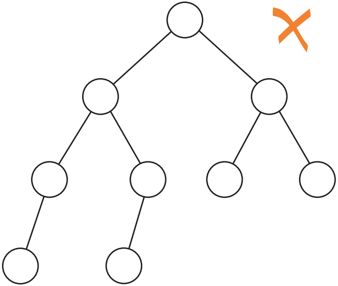




Heaps

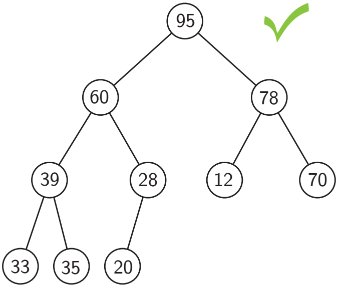
- Árbol binario completo salvo quizás el último nivel, que se llena de izquierda a derecha.

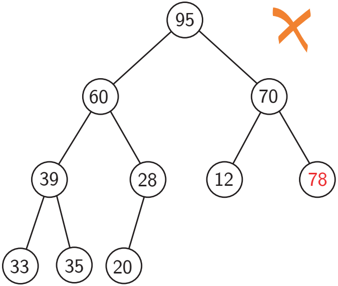




Heaps

- Árbol binario completo salvo quizás el último nivel, que se llena de izquierda a derecha.
- Max-heap: la prioridad de un elemento es mayor o igual a la de sus hijos (por transitividad vale para sus descendientes).



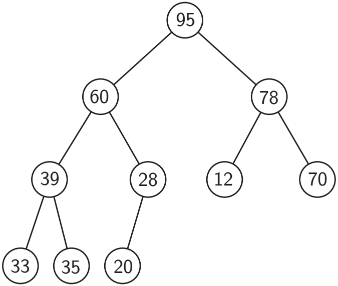


Heaps: representación

Si tenemos un cota para la cantidad de elementos, una representación eficiente es mediante arreglos:

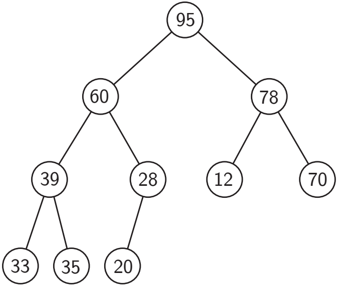
- raíz: $a[1]$
- hijo izquierdo de $a[i]$: $a[2i]$
- hijo derecho de $a[i]$: $a[2i+1]$
- padre de $a[i]$ $(i > 1)$: $a[\lfloor i/2 \rfloor]$

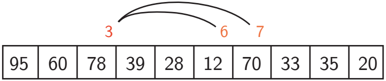
Por la forma del heap, los nodos resultan en n posiciones consecutivas (sin huecos) y la altura del árbol es $O(\log(n))$.



		3			6	7			
95	60	78	39	28	12	70	33	35	20

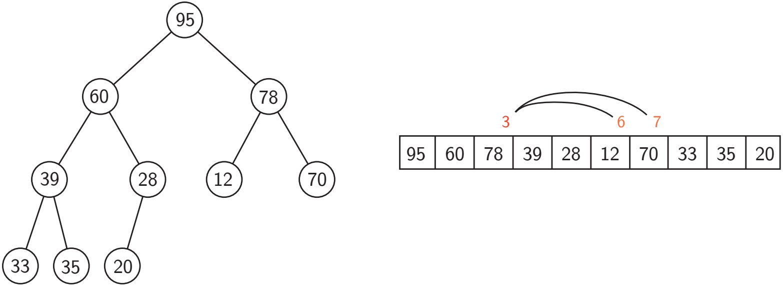
Operaciones: obtener el elemento de máxima prioridad





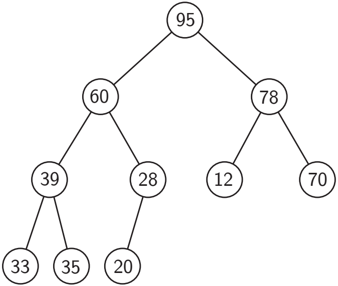
Operaciones: obtener el elemento de máxima prioridad

Por el invariante de representación de un heap, es la raíz (o $a[1]$).
Se obtiene en $O(1)$.



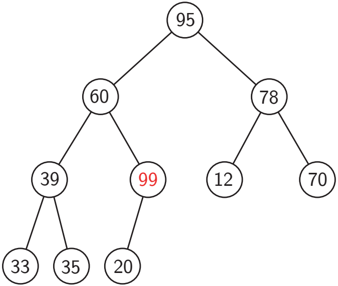
Operaciones: aumentar la prioridad de un elemento

- Para reestablecer el invariante, el elemento eventualmente va a subir.
- Algoritmo: mientras no sea raíz y su prioridad sea mayor que la del padre, intercambiar con el padre.
- Complejidad: $O(\log(n))$.



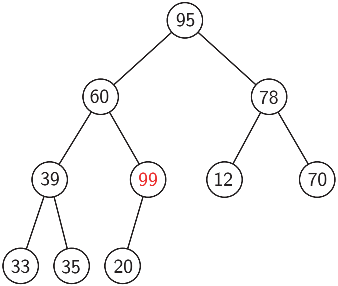
Operaciones: aumentar la prioridad de un elemento

- Para reestablecer el invariante, el elemento eventualmente va a subir.
- Algoritmo: mientras no sea raíz y su prioridad sea mayor que la del padre, intercambiar con el padre.
- Complejidad: $O(\log(n))$.



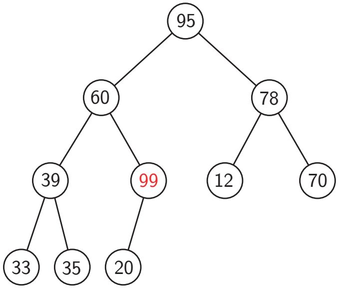
Operaciones: aumentar la prioridad de un elemento

- Para reestablecer el invariante, el elemento eventualmente va a subir.
- Algoritmo: mientras no sea raíz y su prioridad sea mayor que la del padre, intercambiar con el padre.
- Complejidad: $O(\log(n))$.



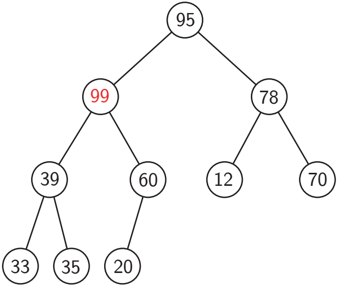
Operaciones: aumentar la prioridad de un elemento

- Para reestablecer el invariante, el elemento eventualmente va a subir.
- Algoritmo: mientras no sea raíz y su prioridad sea mayor que la del padre, intercambiar con el padre.
- Complejidad: $O(\log(n))$.



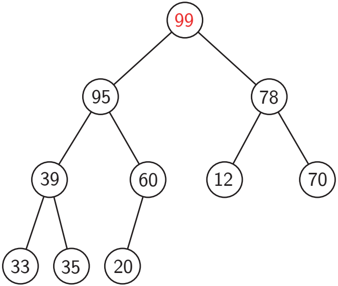
Operaciones: aumentar la prioridad de un elemento

- Para reestablecer el invariante, el elemento eventualmente va a subir.
- Algoritmo: mientras no sea raíz y su prioridad sea mayor que la del padre, intercambiar con el padre.
- Complejidad: $O(\log(n))$.



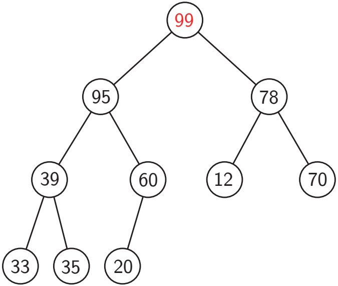
Operaciones: aumentar la prioridad de un elemento

- Para reestablecer el invariante, el elemento eventualmente va a subir.
- Algoritmo: mientras no sea raíz y su prioridad sea mayor que la del padre, intercambiar con el padre.
- Complejidad: $O(\log(n))$.



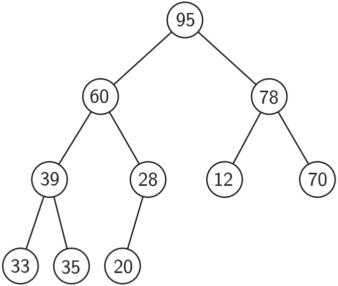
Operaciones: aumentar la prioridad de un elemento

- Para reestablecer el invariante, el elemento eventualmente va a subir.
- Algoritmo: mientras no sea raíz y su prioridad sea mayor que la del padre, intercambiar con el padre.
- Complejidad: $O(\log(n))$.



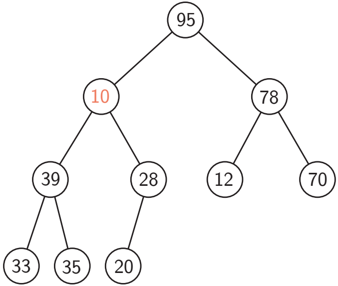
Operaciones: disminuir la prioridad de un elemento

- Para reestablecer el invariante, el elemento eventualmente va a bajar.
- Algoritmo: mientras no sea hoja y su prioridad sea menor que la de alguno de sus hijos, intercambiar con el hijo de mayor prioridad. (algoritmo políticamente incorrecto, a los hijos se los debe querer a todos por igual :))
- Complejidad: $O(\log(n))$.



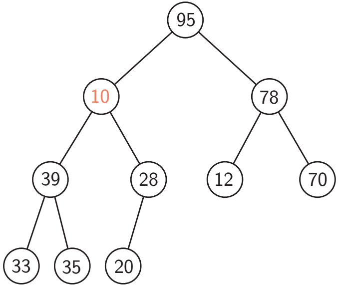
Operaciones: disminuir la prioridad de un elemento

- Para reestablecer el invariante, el elemento eventualmente va a bajar.
- Algoritmo: mientras no sea hoja y su prioridad sea menor que la de alguno de sus hijos, intercambiar con el hijo de mayor prioridad. (algoritmo políticamente incorrecto, a los hijos se los debe querer a todos por igual :))
- Complejidad: $O(\log(n))$.



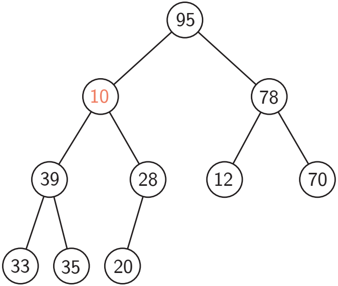
Operaciones: disminuir la prioridad de un elemento

- Para reestablecer el invariante, el elemento eventualmente va a bajar.
- Algoritmo: mientras no sea hoja y su prioridad sea menor que la de alguno de sus hijos, intercambiar con el hijo de mayor prioridad. (algoritmo políticamente incorrecto, a los hijos se los debe querer a todos por igual :))
- Complejidad: $O(\log(n))$.



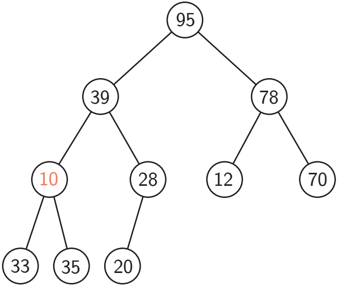
Operaciones: disminuir la prioridad de un elemento

- Para reestablecer el invariante, el elemento eventualmente va a bajar.
- Algoritmo: mientras no sea hoja y su prioridad sea menor que la de alguno de sus hijos, intercambiar con el hijo de mayor prioridad. (algoritmo políticamente incorrecto, a los hijos se los debe querer a todos por igual :))
- Complejidad: $O(\log(n))$.



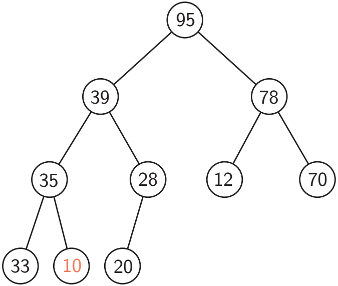
Operaciones: disminuir la prioridad de un elemento

- Para reestablecer el invariante, el elemento eventualmente va a bajar.
- Algoritmo: mientras no sea hoja y su prioridad sea menor que la de alguno de sus hijos, intercambiar con el hijo de mayor prioridad. (algoritmo políticamente incorrecto, a los hijos se los debe querer a todos por igual :))
- Complejidad: $O(\log(n))$.



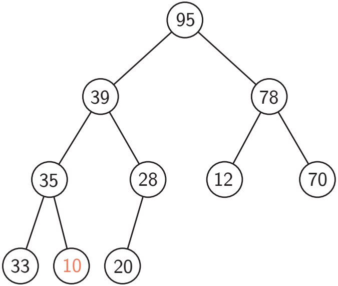
Operaciones: disminuir la prioridad de un elemento

- Para reestablecer el invariante, el elemento eventualmente va a bajar.
- Algoritmo: mientras no sea hoja y su prioridad sea menor que la de alguno de sus hijos, intercambiar con el hijo de mayor prioridad. (algoritmo políticamente incorrecto, a los hijos se los debe querer a todos por igual :))
- Complejidad: $O(\log(n))$.



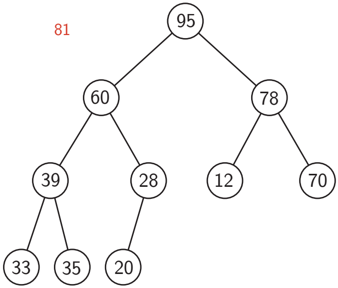
Operaciones: disminuir la prioridad de un elemento

- Para reestablecer el invariante, el elemento eventualmente va a bajar.
- Algoritmo: mientras no sea hoja y su prioridad sea menor que la de alguno de sus hijos, intercambiar con el hijo de mayor prioridad. (algoritmo políticamente incorrecto, a los hijos se los debe querer a todos por igual :))
- Complejidad: $O(\log(n))$.



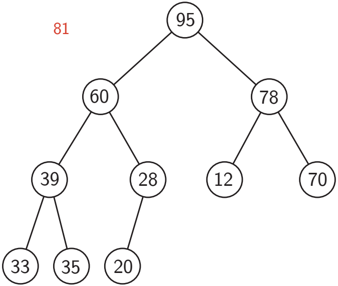
Operaciones: insertar un elemento con una prioridad

- Para mantener la condición de estructura, se agrega al final.
- Para reestablecer el invariante, se asume que el elemento existía con prioridad nula, y se le aumentó la prioridad (algoritmo de subir).
- Complejidad: $O(\log(n))$.



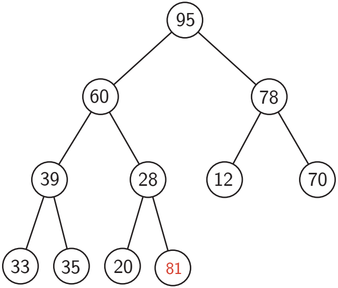
Operaciones: insertar un elemento con una prioridad

- Para mantener la condición de estructura, se agrega al final.
- Para reestablecer el invariante, se asume que el elemento existía con prioridad nula, y se le aumentó la prioridad (algoritmo de subir).
- Complejidad: $O(\log(n))$.



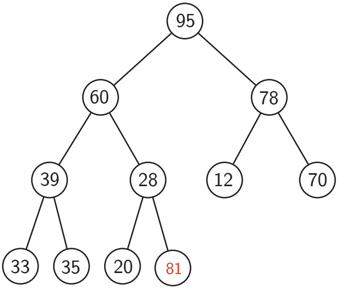
Operaciones: insertar un elemento con una prioridad

- Para mantener la condición de estructura, se agrega al final.
- Para reestablecer el invariante, se asume que el elemento existía con prioridad nula, y se le aumentó la prioridad (algoritmo de subir).
- Complejidad: $O(\log(n))$.



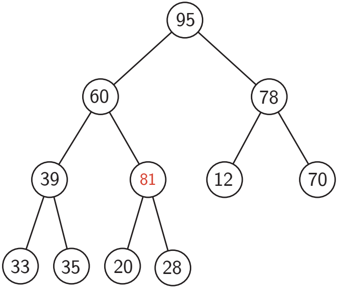
Operaciones: insertar un elemento con una prioridad

- Para mantener la condición de estructura, se agrega al final.
- Para reestablecer el invariante, se asume que el elemento existía con prioridad nula, y se le aumentó la prioridad (algoritmo de subir).
- Complejidad: $O(\log(n))$.



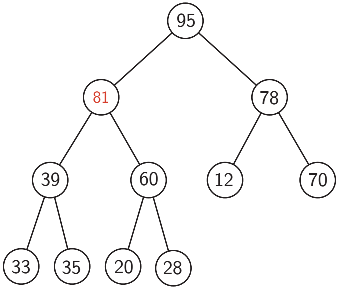
Operaciones: insertar un elemento con una prioridad

- Para mantener la condición de estructura, se agrega al final.
- Para reestablecer el invariante, se asume que el elemento existía con prioridad nula, y se le aumentó la prioridad (algoritmo de subir).
- Complejidad: $O(\log(n))$.



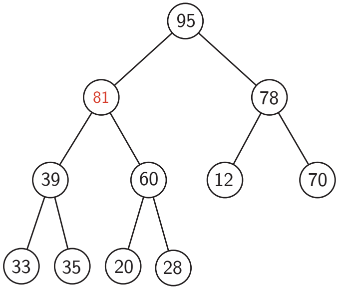
Operaciones: insertar un elemento con una prioridad

- Para mantener la condición de estructura, se agrega al final.
- Para reestablecer el invariante, se asume que el elemento existía con prioridad nula, y se le aumentó la prioridad (algoritmo de subir).
- Complejidad: $O(\log(n))$.



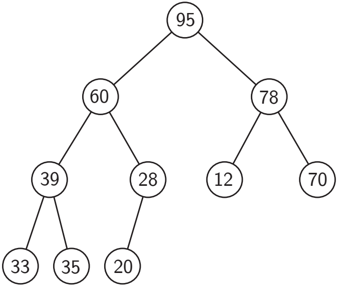
Operaciones: insertar un elemento con una prioridad

- Para mantener la condición de estructura, se agrega al final.
- Para reestablecer el invariante, se asume que el elemento existía con prioridad nula, y se le aumentó la prioridad (algoritmo de subir).
- Complejidad: $O(\log(n))$.



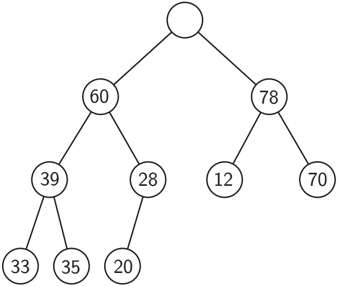
Operaciones: eliminar el elemento de máxima prioridad

- Para mantener la condición de estructura, se elimina la última posición, pasando a la raíz su elemento.
- Para reestablecer el invariante, se asume que lo que se hizo fue disminuir la prioridad de la raíz (algoritmo de bajar).
- Complejidad: $O(\log(n))$.



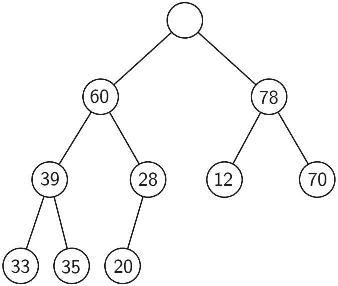
Operaciones: eliminar el elemento de máxima prioridad

- Para mantener la condición de estructura, se elimina la última posición, pasando a la raíz su elemento.
- Para reestablecer el invariante, se asume que lo que se hizo fue disminuir la prioridad de la raíz (algoritmo de bajar).
- Complejidad: $O(\log(n))$.



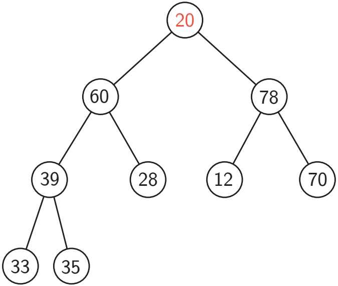
Operaciones: eliminar el elemento de máxima prioridad

- Para mantener la condición de estructura, se elimina la última posición, pasando a la raíz su elemento.
- Para reestablecer el invariante, se asume que lo que se hizo fue disminuir la prioridad de la raíz (algoritmo de bajar).
- Complejidad: $O(\log(n))$.



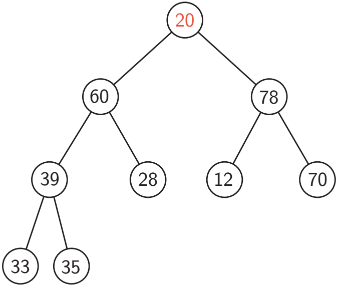
Operaciones: eliminar el elemento de máxima prioridad

- Para mantener la condición de estructura, se elimina la última posición, pasando a la raíz su elemento.
- Para reestablecer el invariante, se asume que lo que se hizo fue disminuir la prioridad de la raíz (algoritmo de bajar).
- Complejidad: $O(\log(n))$.



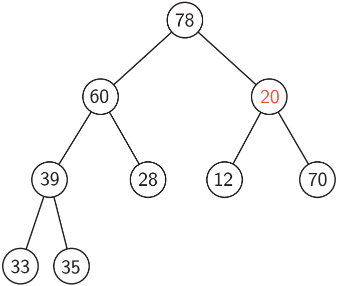
Operaciones: eliminar el elemento de máxima prioridad

- Para mantener la condición de estructura, se elimina la última posición, pasando a la raíz su elemento.
- Para reestablecer el invariante, se asume que lo que se hizo fue disminuir la prioridad de la raíz (algoritmo de bajar).
- Complejidad: $O(\log(n))$.



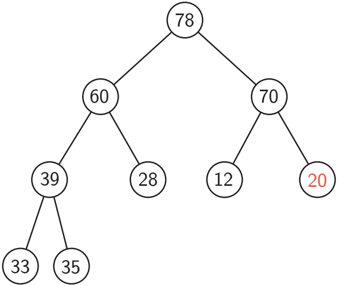
Operaciones: eliminar el elemento de máxima prioridad

- Para mantener la condición de estructura, se elimina la última posición, pasando a la raíz su elemento.
- Para reestablecer el invariante, se asume que lo que se hizo fue disminuir la prioridad de la raíz (algoritmo de bajar).
- Complejidad: $O(\log(n))$.



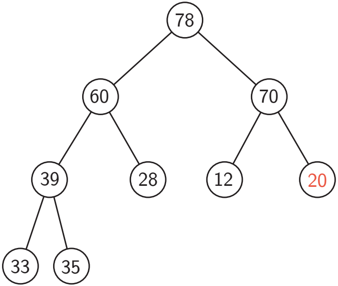
Operaciones: eliminar el elemento de máxima prioridad

- Para mantener la condición de estructura, se elimina la última posición, pasando a la raíz su elemento.
- Para reestablecer el invariante, se asume que lo que se hizo fue disminuir la prioridad de la raíz (algoritmo de bajar).
- Complejidad: $O(\log(n))$.



Operaciones: eliminar el elemento de máxima prioridad

- Para mantener la condición de estructura, se elimina la última posición, pasando a la raíz su elemento.
- Para reestablecer el invariante, se asume que lo que se hizo fue disminuir la prioridad de la raíz (algoritmo de bajar).
- Complejidad: $O(\log(n))$.



Operaciones: construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades

- Si insertamos uno a uno a partir de un heap vacío, la complejidad es $O(\sum_{i=1}^{n} \log(i)) = O(\log(n!)) = O(n \log(n))$ (Teorema de Stirling).
- El algoritmo de Fulkerson, lo que hace es reestablecer sucesivamente el invariante de los subárboles desde las hojas hasta la raíz. Es decir, desde el anteúltimo nivel y subiendo aplica el algoritmo bajar.
- Complejidad: la suma, para cada nodo, de la altura del subárbol que cuelga de él. Se demuestra que es $O(n)$.

Operaciones: construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades

- Si insertamos uno a uno a partir de un heap vacío, la complejidad es $O(\sum_{i=1}^{n} \log(i)) = O(\log(n!)) = O(n \log(n))$ (Teorema de Stirling).
- El algoritmo de Fulkerson, lo que hace es reestablecer sucesivamente el invariante de los subárboles desde las hojas hasta la raíz. Es decir, desde el anteúltimo nivel y subiendo aplica el algoritmo bajar.
- Complejidad: la suma, para cada nodo, de la altura del subárbol que cuelga de él. Se demuestra que es $O(n)$.

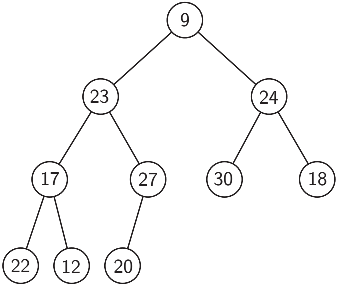
Operaciones: construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades

- Si insertamos uno a uno a partir de un heap vacío, la complejidad es $O(\sum_{i=1}^{n} \log(i)) = O(\log(n!)) = O(n \log(n))$ (Teorema de Stirling).
- El algoritmo de Fulkerson, lo que hace es reestablecer sucesivamente el invariante de los subárboles desde las hojas hasta la raíz. Es decir, desde el anteúltimo nivel y subiendo aplica el algoritmo bajar.
- Complejidad: la suma, para cada nodo, de la altura del subárbol que cuelga de él. Se demuestra que es $O(n)$.

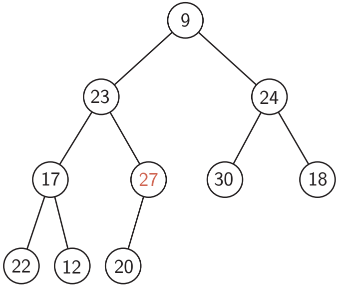
Operaciones: construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades

- Si insertamos uno a uno a partir de un heap vacío, la complejidad es $O(\sum_{i=1}^{n} \log(i)) = O(\log(n!)) = O(n \log(n))$ (Teorema de Stirling).
- El algoritmo de Fulkerson, lo que hace es reestablecer sucesivamente el invariante de los subárboles desde las hojas hasta la raíz. Es decir, desde el anteúltimo nivel y subiendo aplica el algoritmo bajar.
- Complejidad: la suma, para cada nodo, de la altura del subárbol que cuelga de él. Se demuestra que es $O(n)$.

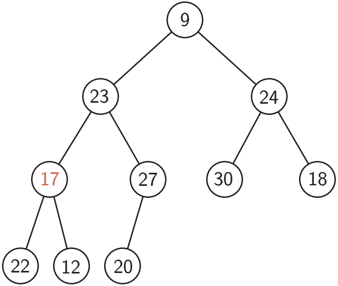
Ejemplo: construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades



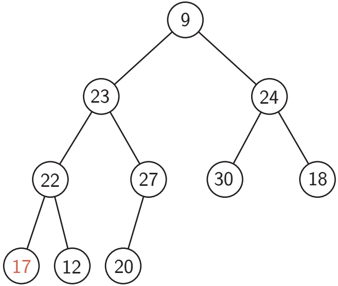
Ejemplo: construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades



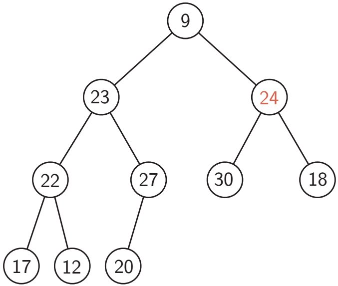
Ejemplo: construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades



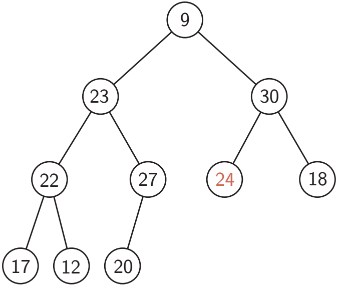
Ejemplo: construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades



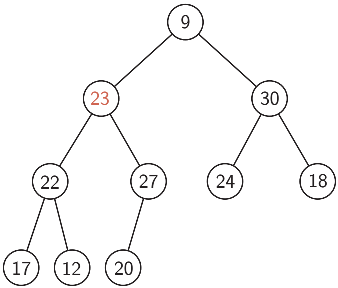
Ejemplo: construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades



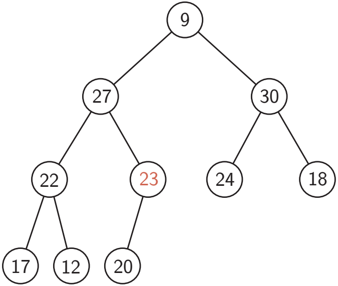
Ejemplo: construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades



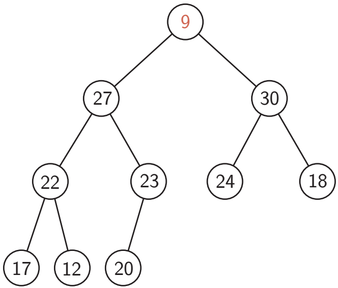
Ejemplo: construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades



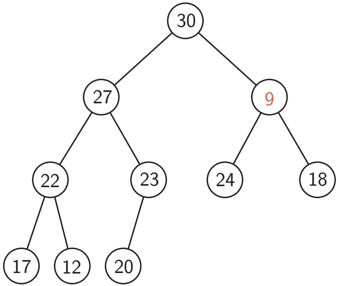
Ejemplo: construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades



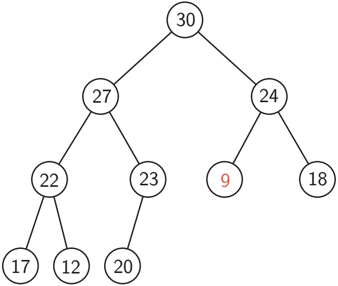
Ejemplo: construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades



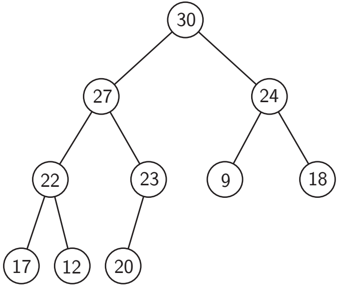
Ejemplo: construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades



Ejemplo: construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades



Ejemplo: construir una cola de prioridad a partir de un conjunto de elementos y sus prioridades



Demostración de la complejidad

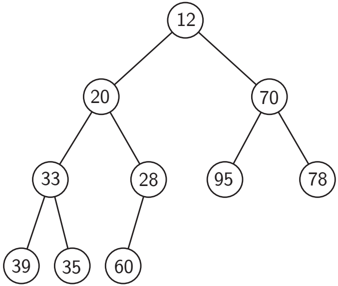
- En el anteúltimo nivel hay a lo sumo $n/2$ nodos, en el anterior $n/4$, y así sucesivamente.... con lo cual, la cantidad de nodos para los cuales el algoritmo de bajar toma en peor caso i pasos es a lo sumo $n/2^i$.
- La complejidad en entonces $\leq \sum_{i=1}^{\log(n)} (n/2^i) i = n \sum_{i=1}^{log(n)} i/2^i \leq 3n/2 = O(n)$ (que la serie $\sum_{i=1}^{\infty} i/2^i$ converge a $3/2$ es un resultado conocido de análisis, y es fácil de ver que converge usando el criterio de D'Alembert).

Demostración de la complejidad

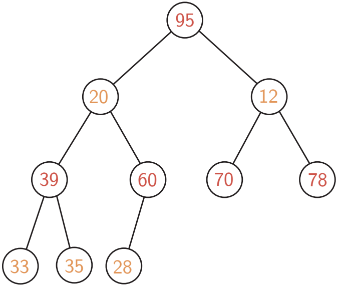
- En el anteúltimo nivel hay a lo sumo $n/2$ nodos, en el anterior $n/4$, y así sucesivamente.... con lo cual, la cantidad de nodos para los cuales el algoritmo de bajar toma en peor caso i pasos es a lo sumo $n/2^i$.
- La complejidad en entonces $\leq \sum_{i=1}^{\log(n)} (n/2^i)i = n \sum_{i=1}^{log(n)} i/2^i \leq 3n/2 = O(n)$ (que la serie $\sum_{i=1}^{\infty} i/2^i$ converge a $3/2$ es un resultado conocido de análisis, y es fácil de ver que converge usando el criterio de D'Alembert).

Heaps: variantes

- Min-heap: la prioridad de un elemento es menor o igual a la de sus hijos (por transitividad vale para sus descendientes).
- Max-min-heap: la prioridad de un elemento es mayor (resp. menor) o igual a la de sus descendientes si está en un nivel par (resp. impar).



Min-heap



Max-min-heap

Heaps: variantes

Variantes más avanzadas (que permiten la unión eficiente de dos heaps en un nuevo heap):

- Heaps izquierdistas
- Heaps binomiales
- Heaps de Fibonacci

Lo que cambia es la propiedad estructural (no son ya "árboles completos salvo quizás el último nivel, que se llena de izquierda a derecha"), pero se mantiene la propiedad de orden.