Colas de prioridad y heaps

Nicolás D'Ippolito[1,2], Ariel Bendersky[1,2]

[1]Instituto de Investigaciones en Ciencias de la Computación - ICC, CONICET, Argentina.
[2]Departamento de Computación, FCEyN, Universidad de Buenos Aires, Buenos Aires, Argentina.

Algoritmos y Estructuras de Datos II
Segundo cuatrimestre de 2019

(2) Colas de prioridad

- Muchas aplicaciones.

(2) Colas de prioridad

- Muchas aplicaciones.
    - Sistemas operativos.

(2) Colas de prioridad

- Muchas aplicaciones.
  - Sistemas operativos.
  - Algoritmos de scheduling.

- Muchas aplicaciones.
  - Sistemas operativos.
  - Algoritmos de scheduling.
  - Gestión de colas en cualquier ambiente, etc.

(2) Colas de prioridad

- Muchas aplicaciones.
  - Sistemas operativos.
  - Algoritmos de scheduling.
  - Gestión de colas en cualquier ambiente, etc.
- La prioridad, en general, la epresamos con un entero, pero puede ser cualquier tipo α con un orden $<_\alpha$ asociado.

(2) Colas de prioridad

- Muchas aplicaciones.
  - Sistemas operativos.
  - Algoritmos de scheduling.
  - Gestión de colas en cualquier ambiente, etc.
- La prioridad, en general, la epresamos con un entero, pero puede ser cualquier tipo α con un orden $<_\alpha$ asociado.
- Hay una correspondencia entre la máxima prioridad y el valor máximo (o mínimo) del tipo α.

(3) El TAD cola de prioridad

TAD $\textsc{ColaPrior}(\alpha, <_\alpha)$

observadores básicos

vacía? : ColaPrior(α, $<_\alpha$) $\longrightarrow$ bool

próximo : ColaPrior(α, $<_\alpha$)*c* $\longrightarrow$ α $\{\neg$vacía(α)$\}$

desencolar : ColaPrior(α, $<_\alpha$)*c* $\longrightarrow$ ColaPrior(α, $<_\alpha$)

$\{\neg$vacía(α)$\}$

generadores

vacía : $\longrightarrow$ ColaPrior(α, $<_\alpha$)

encolar : α $\times$ ColaPrior(α, $<_\alpha$) $\longrightarrow$ ColaPrior(α, $<_\alpha$)

otras operaciones

$\bullet =_{\text{colaPrior}} \bullet$: ColaPrior(α, $<_\alpha$) $\times$ ColaPrior(α, $<_\alpha$) $\longrightarrow$ bool

Fin TAD

(4) Representación de colas de prioridad

- La implementación más eficiente es a través de *heaps*.

(4) Representación de colas de prioridad

- La implementación más eficiente es a través de *heaps*.
- Heap significa montículo o montón.

(5) Representación de colas de prioridad

- ColaPrior(α, $<_\alpha$) se representa con un heap.

(5) Representación de colas de prioridad

- ColaPrior(α, $<_\alpha$) se representa con un heap.
- Un heap es una estructura con el siguiente invariante de representación (Condición de heap):

(5) Representación de colas de prioridad

- ColaPrior(α, $<_\alpha$) se representa con un heap.
- Un heap es una estructura con el siguiente invariante de representación (Condición de heap):
  - Árbol binario perfectamente balanceado.

(5) Representación de colas de prioridad

- ColaPrior(α, $<_\alpha$) se representa con un heap.
- Un heap es una estructura con el siguiente invariante de representación (Condición de heap):
  - Árbol binario perfectamente balanceado.
  - La clave (prioridad) de cada nodo es mayor o igual que la de sus hijos (si los tiene).

(5) Representación de colas de prioridad

- ColaPrior(α, $<_\alpha$) se representa con un heap.
- Un heap es una estructura con el siguiente invariante de representación (Condición de heap):
  - Árbol binario perfectamente balanceado.
  - La clave (prioridad) de cada nodo es mayor o igual que la de sus hijos (si los tiene).
  - Todo subárbol es un heap.

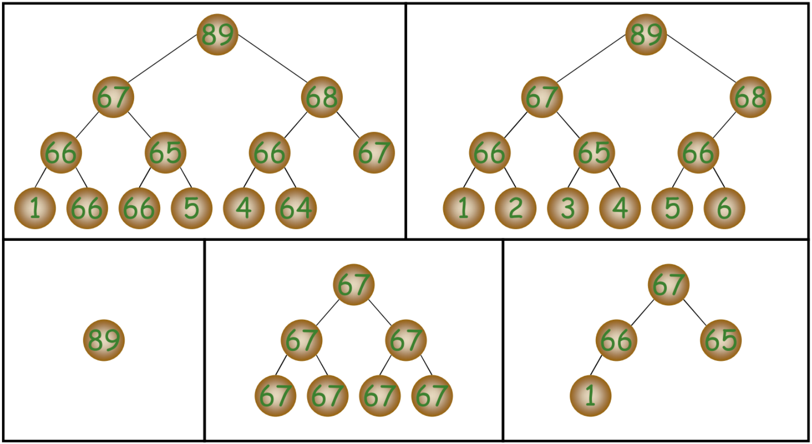
(5) Representación de colas de prioridad

- ColaPrior(α, $<_\alpha$) se representa con un heap.
- Un heap es una estructura con el siguiente invariante de representación (Condición de heap):
  - Árbol binario perfectamente balanceado.
  - La clave (prioridad) de cada nodo es mayor o igual que la de sus hijos (si los tiene).
  - Todo subárbol es un heap.
  - (No obligatorio) Es izquierdiza. Es decir, el último nivel está llendo desde la izquierda.

(5) Representación de colas de prioridad

- ColaPrior(α, $<_\alpha$) se representa con un heap.
- Un heap es una estructura con el siguiente invariante de representación (Condición de heap):
  - Árbol binario perfectamente balanceado.
  - La clave (prioridad) de cada nodo es mayor o igual que la de sus hijos (si los tiene).
  - Todo subárbol es un heap.
  - (No obligatorio) Es izquierdiza. Es decir, el último nivel está llendo desde la izquierda.
- ¡Ojo! No es un ABB ni una estruactura totalmente ordenada.

(6) ¿Cuáles son heaps?



(7) max-heaps vs. min-heaps

- Esta estructura, con el máximo arriba, se llama max-heap.

(7) max-heaps vs. min-heaps

- Esta estructura, con el máximo arriba, se llama max-heap.
- Se puede invertir el signo de la comparación y armar un min-heap.

(7) max-heaps vs. min-heaps

- Esta estructura, con el máximo arriba, se llama max-heap.
- Se puede invertir el signo de la comparación y armar un min-heap.
- Incluso, se pueden ordenar datos que no son numéricos, siempre y cuando se defina la comparación $<_\alpha$.

(8) Operaciones sobre un heap

- Tenemos definidas las mismas operaciónes que en el TAD *Cola de Prioridad*.

(8) Operaciones sobre un heap

- Tenemos definidas las mismas operaciónes que en el TAD *Cola de Prioridad*.
  - Vacío: crea un heap vacío.

(8) Operaciones sobre un heap

- Tenemos definidas las mismas operaciónes que en el TAD *Cola de Prioridad*.
  - Vacío: crea un heap vacío.
  - Próximo: devuelve el elemento de máxima prioridad sin modificar el heap.

(8) Operaciones sobre un heap

- Tenemos definidas las mismas operaciónes que en el TAD *Cola de Prioridad*.
  - Vacío: crea un heap vacío.
  - Próximo: devuelve el elemento de máxima prioridad sin modificar el heap.
  - Encolar: agrega un nuevo elemento, preservando el invariante del heap.

(8) Operaciones sobre un heap

- Tenemos definidas las mismas operaciónes que en el TAD *Cola de Prioridad*.
  - Vacío: crea un heap vacío.
  - Próximo: devuelve el elemento de máxima prioridad sin modificar el heap.
  - Encolar: agrega un nuevo elemento, preservando el invariante del heap.
  - Desencolar: elimina el elemento de máxima prioridad y restablece el invariante.

(9) Implementación de heaps

- Todas las representaciones usadas para árboles binarios son admisibles.

(9) Implementación de heaps

- Todas las representaciones usadas para árboles binarios son admisibles.
  - Representación con punteros. De ser necesario también con punteros hijo-padre.

- Todas las representaciones usadas para árboles binarios son admisibles.
  - Representación con punteros. De ser necesario también con punteros hijo-padre.
  - Representación con arrays. Es particularmente eficiente.

(10) Representación con arrays

- Cada nodo v es almacenado en la posición $p(v)$.

- Cada nodo v es almacenado en la posición $p(v)$.
- Si v es la raíz, entonces $p(v) = 0$.

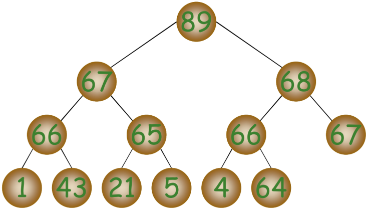
(10) Representación con arrays

- Cada nodo v es almacenado en la posición $p(v)$.
- Si v es la raíz, entonces $p(v) = 0$.
- Si v es el hijo izquierdo de u entonces $p(v) = 2p(u) + 1$.

(10) Representación con arrays

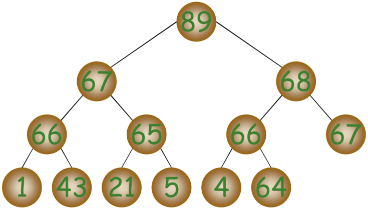
- Cada nodo v es almacenado en la posición $p(v)$.
- Si v es la raíz, entonces $p(v) = 0$.
- Si v es el hijo izquierdo de u entonces $p(v) = 2p(u) + 1$.
- Si v es el hijo derecho de u entonces $p(v) = 2p(u) + 2$.

- Cada nodo v es almacenado en la posición $p(v)$.
- Si v es la raíz, entonces $p(v) = 0$.
- Si v es el hijo izquierdo de u entonces $p(v) = 2p(u) + 1$.
- Si v es el hijo derecho de u entonces $p(v) = 2p(u) + 2$.
- El siguiente heap



(10) Representación con arrays

- Cada nodo v es almacenado en la posición $p(v)$.
- Si v es la raíz, entonces $p(v) = 0$.
- Si v es el hijo izquierdo de u entonces $p(v) = 2p(u) + 1$.
- Si v es el hijo derecho de u entonces $p(v) = 2p(u) + 2$.
- El siguiente heap



- Se representa con $[89, 67, 68, 66, 65, 66, 67, 1, 43, 21, 5, 4, 64]$.

(11) Heaps sobre arrays

- Ventajas:

- Ventajas:
  - Muy eficientes en términos de espacio.

- Ventajas:
  - Muy eficientes en términos de espacio.
  - Muy fáciles de navegar:

- Ventajas:
  - Muy eficientes en términos de espacio.
  - Muy fáciles de navegar:
    - Si i es el padre y j_{izq} y j_{der} los hijos, entonces $j_{izq} = 2i + 1$ y $j_{der} = 2i + 2$.

- Ventajas:
  - Muy eficientes en términos de espacio.
  - Muy fáciles de navegar:
    - Si i es el padre y j_{izq} y j_{der} los hijos, entonces $j_{izq} = 2i + 1$ y $j_{der} = 2i + 2$.
    - Si i es el hijo, su padre j es tal que $j = \left\lfloor \frac{i-1}{2} \right\rfloor$.

(11) Heaps sobre arrays

- Ventajas:
  - Muy eficientes en términos de espacio.
  - Muy fáciles de navegar:
    - Si i es el padre y j_{izq} y j_{der} los hijos, entonces $j_{izq} = 2i + 1$ y $j_{der} = 2i + 2$.
    - Si i es el hijo, su padre j es tal que $j = \left\lfloor \frac{i-1}{2} \right\rfloor$.
- Desventaja:

(11) Heaps sobre arrays

- Ventajas:
  - Muy eficientes en términos de espacio.
  - Muy fáciles de navegar:
    - Si i es el padre y j_{izq} y j_{der} los hijos, entonces $j_{izq} = 2i + 1$ y $j_{der} = 2i + 2$.
    - Si i es el hijo, su padre j es tal que $j = \left\lfloor \frac{i-1}{2} \right\rfloor$.
- Desventaja:
  - Es una implementación estática. Si necesitamos agregar más elementos que los que reservamos, necesitamos duplicar el arreglo.

(12) Algoritmo para *Próximo*

- El elemento de prioridad máxima está en la posición 0 del arreglo.

(12) Algoritmo para *Próximo*

- El elemento de prioridad máxima está en la posición 0 del arreglo.
- La operación tiene costo constante $O(1)$.

(13) Algoritmo para *Encolar*

- encolar(elemento)

(13) Algoritmo para *Encolar*

- encolar(elemento)
  - insertarAlFinal(elemento)

(13) Algoritmo para *Encolar*

- encolar(elemento)
    - insertarAlFinal(elemento)
    - percolarHaciaArriba(elemento)

(13) Algoritmo para *Encolar*

- encolar(elemento)
  - insertarAlFinal(elemento)
  - percolarHaciaArriba(elemento)
- percolarHaciaArriba(elemento)

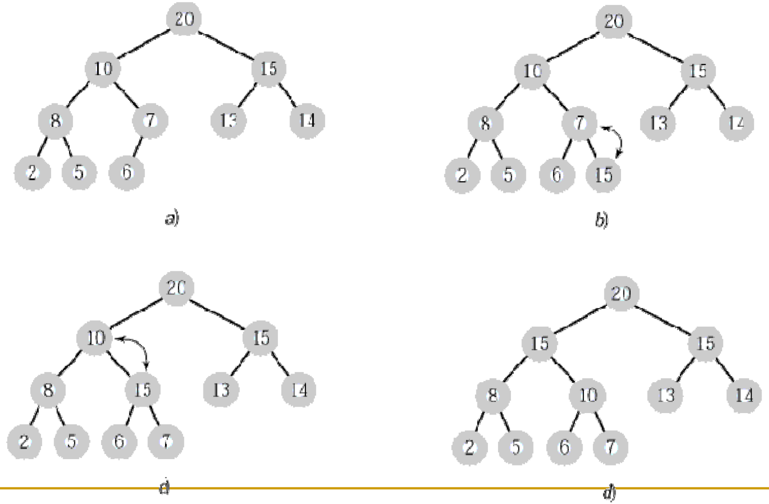
(13) Algoritmo para *Encolar*

- encolar(elemento)
  - insertarAlFinal(elemento)
  - percolarHaciaArriba(elemento)
- percolarHaciaArriba(elemento)
  - while (elemento no es raíz) $\wedge_L$ (prioridad(elemento)$>$prioridad(padre(elemento)))

(13) Algoritmo para *Encolar*

- encolar(elemento)
  - insertarAlFinal(elemento)
  - percolarHaciaArriba(elemento)
- percolarHaciaArriba(elemento)
  - while (elemento no es raíz) $\wedge_L$ (prioridad(elemento)$>$prioridad(padre(elemento)))
    - intercambiar el elemento con el padre

(14) Ejemplo para *Encolar*



(15) Algoritmo para *Desencolar*

- Desencolar

(15) Algoritmo para *Desencolar*

- Desencolar
  - Reemplazar el primer elemento con la última hoja y eliminar la última hoja.

(15) Algoritmo para *Desencolar*

- Desencolar
  - Reemplazar el primer elemento con la última hoja y eliminar la última hoja.
  - percolarHaciaAbajo(raíz)

(15) Algoritmo para *Desencolar*

- Desencolar
  - Reemplazar el primer elemento con la última hoja y eliminar la última hoja.
  - percolarHaciaAbajo(raíz)
- percolarHaciaAbajo(p)

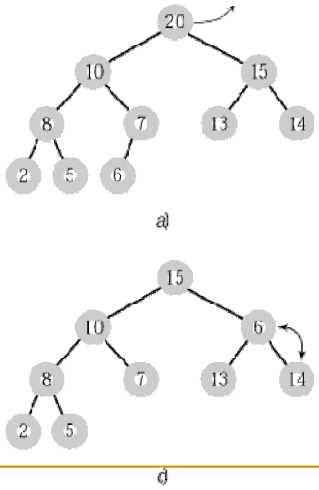
(15) Algoritmo para *Desencolar*

- Desencolar
  - Reemplazar el primer elemento con la última hoja y eliminar la última hoja.
  - percolarHaciaAbajo(raíz)
- percolarHaciaAbajo(p)
  - while (p no es hoja) $\wedge_L$ (prioridad(p)<(prioridad(algún hijo de p)))

(15) Algoritmo para *Desencolar*

- Desencolar
  - Reemplazar el primer elemento con la última hoja y eliminar la última hoja.
  - percolarHaciaAbajo(raíz)
- percolarHaciaAbajo(p)
  - while (p no es hoja) $\wedge_L$ (prioridad(p)<(prioridad(algún hijo de p)))
    - intercambiar p con el hijo de mayor prioridad

(16) Ejemplo para *Desencolar*



(17) costos

- Encolar y desencolar son proporcionales a la altura del heap. Es decir, tardan $O(\log(n))$.

(18) Heapify

- Dado un array arr, lo queremos transformar en un heap a través de la permutación de sus elementos.

(18) Heapify

- Dado un array arr, lo queremos transformar en un heap a través de la permutación de sus elementos.
- Algoritmo simple:

(18) Heapify

- Dado un array arr, lo queremos transformar en un heap a través de la permutación de sus elementos.
- Algoritmo simple:
  - Para i desde 1 hasta tam(arr)

(18) Heapify

- Dado un array arr, lo queremos transformar en un heap a través de la permutación de sus elementos.
- Algoritmo simple:
  - Para i desde 1 hasta tam(arr)
    - encolar(arr[i])

(18) Heapify

- Dado un array arr, lo queremos transformar en un heap a través de la permutación de sus elementos.
- Algoritmo simple:
  - Para i desde 1 hasta tam(arr)
    - encolar(arr[i])
- El tiempo que tarda es:

$$\sum_{j=1}^{n} \log(j) = \log(n!) = \Theta(n \log(n))$$

donde usamos la aproximación de Stirling.

(18) Heapify

- Dado un array arr, lo queremos transformar en un heap a través de la permutación de sus elementos.
- Algoritmo simple:
  - Para i desde 1 hasta tam(arr)
    - encolar(arr[i])
- El tiempo que tarda es:

$$\sum_{j=1}^{n} \log(j) = \log(n!) = \Theta(n \log(n))$$

  donde usamos la aproximación de Stirling.
- ¿Se puede hacer más rápido?

- Dado un array arr, lo queremos transformar en un heap a través de la permutación de sus elementos.
- Algoritmo simple:
    - Para i desde 1 hasta tam(arr)
        - encolar(arr[i])
- El tiempo que tarda es:

$$\sum_{j=1}^{n} \log(j) = \log(n!) = \Theta(n \log(n))$$

  donde usamos la aproximación de Stirling.
- ¿Se puede hacer más rápido?

(18) Heapify

- Dado un array arr, lo queremos transformar en un heap a través de la permutación de sus elementos.
- Algoritmo simple:
  - Para i desde 1 hasta tam(arr)
    - encolar(arr[i])
- El tiempo que tarda es:

$$\sum_{j=1}^{n} \log(j) = \log(n!) = \Theta(n \log(n))$$

  donde usamos la aproximación de Stirling.
- ¿Se puede hacer más rápido? Floyd mostró que sí.

(19) Heapify - Algoritmo de Floyd

- El algoritmo de Floyd se basa en la aplicación de la función percolarHaciaAbajo a árboles binarios cuyos subárboles derecho e izquierdo ya cumplen la condición de heap.

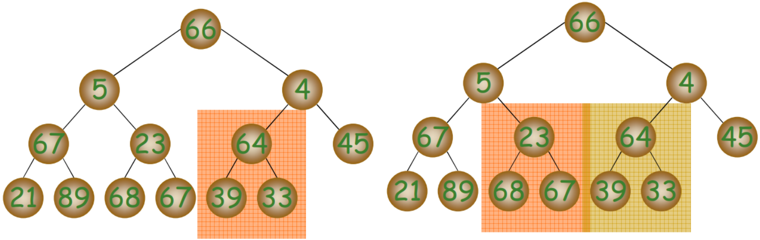
(19) Heapify - Algoritmo de Floyd

- El algoritmo de Floyd se basa en la aplicación de la función percolarHaciaAbajo a árboles binarios cuyos subárboles derecho e izquierdo ya cumplen la condición de heap.
- Prograsivamente se *heapifican* los subárboles cuya raíz está en el penúltimo nivel, luego los del nivel anterior, y así sucesivamente hasta llegar a la raíz del árbol.

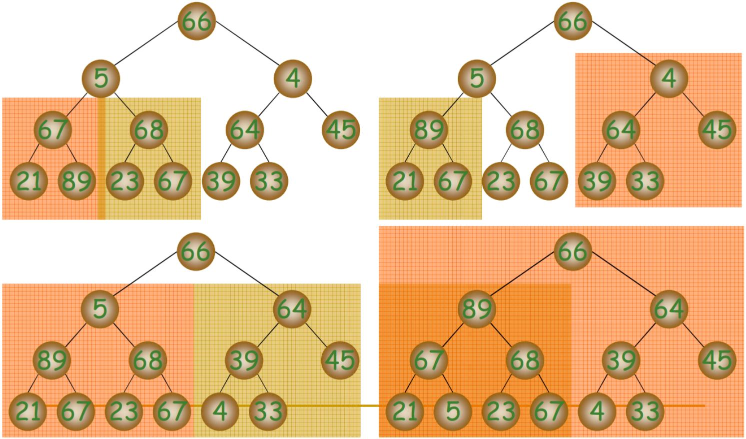
(19) Heapify - Algoritmo de Floyd

- El algoritmo de Floyd se basa en la aplicación de la función percolarHaciaAbajo a árboles binarios cuyos subárboles derecho e izquierdo ya cumplen la condición de heap.
- Prograsivamente se *heapifican* los subárboles cuya raíz está en el penúltimo nivel, luego los del nivel anterior, y así sucesivamente hasta llegar a la raíz del árbol.
- Es una estrategia bottom-up.

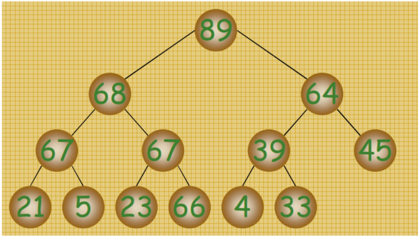
0	1	2	3	4	5	6	7	8	9	10	11	12
66	5	4	67	23	64	45	21	89	68	67	39	33



(21) Algoritmo de Floyd - Ejemplo



(22) Algoritmo de Floyd - Ejemplo



0	1	2	3	4	5	6	7	8	9	10	11	12
89	68	64	67	67	39	45	21	5	23	66	4	33

(23) Complejidad del algoritmo de Floyd

- Notemos que en un árbol binario perfectamente balanceado, el último nivel tiene a lo sumo $n/2$ nodos. El anterior tiene exactamente $n/4$ nodos. Luego $n/8$. Hasta llegar al primero que tiene un solo nodo.

(23) Complejidad del algoritmo de Floyd

- Notemos que en un árbol binario perfectamente balanceado, el último nivel tiene a lo sumo $n/2$ nodos. El anterior tiene exactamente $n/4$ nodos. Luego $n/8$. Hasta llegar al primero que tiene un solo nodo.
- Luego, $n/4$ nodos tendrán que percolar a lo sumo una vez. $n/8$ tendrán que percolar a lo sumo dos veces, y así sucesivamente.

(23) Complejidad del algoritmo de Floyd

- Notemos que en un árbol binario perfectamente balanceado, el último nivel tiene a lo sumo $n/2$ nodos. El anterior tiene exactamente $n/4$ nodos. Luego $n/8$. Hasta llegar al primero que tiene un solo nodo.
- Luego, $n/4$ nodos tendrán que percolar a lo sumo una vez. $n/8$ tendrán que percolar a lo sumo dos veces, y así sucesivamente.
- La cantidad total de intercambios es entonces, en el peor caso:

$$1\frac{n}{4} + 2\frac{n}{8} + 3\frac{n}{16} + \ldots = \sum_{i=1}^{\log(n)} i\frac{n}{2^{i+1}} = \frac{n}{2}\sum_{i=1}^{\log(n)} \frac{i}{2^i} \leq \frac{n}{2}\sum_{i=1}^{\infty} \frac{i}{2^i}$$

Y como esa última suma es convergente, es $O(n)$.

- Implementación de operaciones no estándar de heaps:

- Implementación de operaciones no estándar de heaps:
  - Eliminación de una clave cualquiera.

(24) Aplicaciones del algoritmo de Floyd

- Implementación de operaciones no estándar de heaps:
  - Eliminación de una clave cualquiera.
  - Ejemplo: kill de un proceso dado su PID.

(24) Aplicaciones del algoritmo de Floyd

- Implementación de operaciones no estándar de heaps:
  - Eliminación de una clave cualquiera.
  - Ejemplo: kill de un proceso dado su PID.
  - Lleva $O(n)$ encontrar la clave, y $O(n)$ reconstruir el invariante de representación del heap con el algoritmo de Floyd.

(24) Aplicaciones del algoritmo de Floyd

- Implementación de operaciones no estándar de heaps:
  - Eliminación de una clave cualquiera.
  - Ejemplo: kill de un proceso dado su PID.
  - Lleva $O(n)$ encontrar la clave, y $O(n)$ reconstruir el invariante de representación del heap con el algoritmo de Floyd.
- Sirve para ordenar en tiempo $\Theta(n \log(n))$.

(25) HeapSort

- ¿Podemos ordenar con un heap?

(25) HeapSort

- ¿Podemos ordenar con un heap?
- ¿Cómo sería?

(25) HeapSort

- ¿Podemos ordenar con un heap?
- ¿Cómo sería?
- Se puede hacer algo todavía mejor, *sin memoria adicional*.

(25) HeapSort

- ¿Podemos ordenar con un heap?
- ¿Cómo sería?
- Se puede hacer algo todavía mejor, *sin memoria adicional*.
- Usamos el algoritmo de Floyd Complejidad: $O(n)$

(25) HeapSort

- ¿Podemos ordenar con un heap?
- ¿Cómo sería?
- Se puede hacer algo todavía mejor, *sin memoria adicional*.
- Usamos el algoritmo de Floyd Complejidad: $O(n)$
- Algoritmo:

(25) HeapSort

- ¿Podemos ordenar con un heap?
- ¿Cómo sería?
- Se puede hacer algo todavía mejor, *sin memoria adicional*.
- Usamos el algoritmo de Floyd Complejidad: $O(n)$
- Algoritmo:

  heapify(A, 0, $n-1$)

- ¿Podemos ordenar con un heap?
- ¿Cómo sería?
- Se puede hacer algo todavía mejor, *sin memoria adicional*.
- Usamos el algoritmo de Floyd Complejidad: $O(n)$
- Algoritmo:

  heapify(A, 0, $n-1$)

  Recorro con i desde $n-1$ a 0:

(25) HeapSort

- ¿Podemos ordenar con un heap?
- ¿Cómo sería?
- Se puede hacer algo todavía mejor, *sin memoria adicional*.
- Usamos el algoritmo de Floyd Complejidad: $O(n)$
- Algoritmo:

  heapify(A, 0, $n-1$)

  Recorro con i desde $n-1$ a 0:

  // A[0] es el máximo elemento, lo pongo al final.

(25) HeapSort

- ¿Podemos ordenar con un heap?
- ¿Cómo sería?
- Se puede hacer algo todavía mejor, *sin memoria adicional*.
- Usamos el algoritmo de Floyd Complejidad: $O(n)$
- Algoritmo:

  heapify(A, 0, $n-1$)

  Recorro con i desde $n-1$ a 0:

      // A[0] es el máximo elemento, lo pongo al final.
      max = A[0]

(25) HeapSort

- ¿Podemos ordenar con un heap?
- ¿Cómo sería?
- Se puede hacer algo todavía mejor, *sin memoria adicional*.
- Usamos el algoritmo de Floyd Complejidad: $O(n)$
- Algoritmo:

  heapify(A, 0, $n-1$)

  Recorro con i desde $n-1$ a 0:

  > // A[0] es el máximo elemento, lo pongo al final.
  > max = A[0]
  > desencolar(A, 0, i) // es la operación de heap sobre arreglo.

(25) HeapSort

- ¿Podemos ordenar con un heap?
- ¿Cómo sería?
- Se puede hacer algo todavía mejor, *sin memoria adicional*.
- Usamos el algoritmo de Floyd Complejidad: $O(n)$
- Algoritmo:

  heapify(A, 0, $n-1$)

  Recorro con i desde $n-1$ a 0:

  // A[0] es el máximo elemento, lo pongo al final.
  max = A[0]
  desencolar(A, 0, i) // es la operación de heap sobre arreglo.
  A[i] = max

(25) HeapSort

- ¿Podemos ordenar con un heap?
- ¿Cómo sería?
- Se puede hacer algo todavía mejor, *sin memoria adicional*.
- Usamos el algoritmo de Floyd Complejidad: $O(n)$
- Algoritmo:

  heapify(A, 0, $n-1$)

  Recorro con i desde $n-1$ a 0:

  // A[0] es el máximo elemento, lo pongo al final.
  max = A[0]
  desencolar(A, 0, i) // es la operación de heap sobre arreglo.
  A[i] = max
- ¿Cuál es el invariante?

(25) HeapSort

- ¿Podemos ordenar con un heap?
- ¿Cómo sería?
- Se puede hacer algo todavía mejor, *sin memoria adicional*.
- Usamos el algoritmo de Floyd Complejidad: $O(n)$
- Algoritmo:

  heapify(A, 0, $n-1$)

  Recorro con i desde $n-1$ a 0:

  // A[0] es el máximo elemento, lo pongo al final.
  max = A[0]
  desencolar(A, 0, i) // es la operación de heap sobre arreglo.
  A[i] = max
- ¿Cuál es el invariante?
  - $A[i \dots n-1]$ está ordenado,

(25) HeapSort

- ¿Podemos ordenar con un heap?
- ¿Cómo sería?
- Se puede hacer algo todavía mejor, *sin memoria adicional*.
- Usamos el algoritmo de Floyd Complejidad: $O(n)$
- Algoritmo:

  heapify(A, 0, $n-1$)

  Recorro con i desde $n-1$ a 0:

  // A[0] es el máximo elemento, lo pongo al final.
  max = A[0]
  desencolar(A, 0, i) // es la operación de heap sobre arreglo.
  A[i] = max
- ¿Cuál es el invariante?
  - $A[i \dots n-1]$ está ordenado,
  - $A[0 \dots i-1]$ es un heap

(25) HeapSort

- ¿Podemos ordenar con un heap?
- ¿Cómo sería?
- Se puede hacer algo todavía mejor, *sin memoria adicional*.
- Usamos el algoritmo de Floyd Complejidad: $O(n)$
- Algoritmo:

  heapify(A, 0, $n-1$)

  Recorro con i desde $n-1$ a 0:

  // A[0] es el máximo elemento, lo pongo al final.
  max = A[0]
  desencolar(A, 0, i) // es la operación de heap sobre arreglo.
  A[i] = max

- ¿Cuál es el invariante?
  - $A[i \dots n-1]$ está ordenado,
  - $A[0 \dots i-1]$ es un heap
- ¿Cuál es la complejidad?

(25) HeapSort

- ¿Podemos ordenar con un heap?
- ¿Cómo sería?
- Se puede hacer algo todavía mejor, *sin memoria adicional*.
- Usamos el algoritmo de Floyd Complejidad: $O(n)$
- Algoritmo:

  heapify(A, 0, $n-1$)

  Recorro con i desde $n-1$ a 0:

  // A[0] es el máximo elemento, lo pongo al final.
  max = A[0]
  desencolar(A, 0, i) // es la operación de heap sobre arreglo.
  A[i] = max

- ¿Cuál es el invariante?
  - $A[i \dots n-1]$ está ordenado,
  - $A[0 \dots i-1]$ es un heap
- ¿Cuál es la complejidad?

- ¿Podemos ordenar con un heap?
- ¿Cómo sería?
- Se puede hacer algo todavía mejor, *sin memoria adicional*.
- Usamos el algoritmo de Floyd Complejidad: $O(n)$
- Algoritmo:

  heapify(A, 0, $n-1$)

  Recorro con i desde $n-1$ a 0:

  // A[0] es el máximo elemento, lo pongo al final.
  max = A[0]
  desencolar(A, 0, i) // es la operación de heap sobre arreglo.
  A[i] = max
- ¿Cuál es el invariante?
  - $A[i \ldots n-1]$ está ordenado,
  - $A[0 \ldots i-1]$ es un heap
- ¿Cuál es la complejidad? $O(n)+$

(25) HeapSort

- ¿Podemos ordenar con un heap?
- ¿Cómo sería?
- Se puede hacer algo todavía mejor, *sin memoria adicional*.
- Usamos el algoritmo de Floyd Complejidad: $O(n)$
- Algoritmo:

  heapify(A, 0, $n-1$)

  Recorro con i desde $n-1$ a 0:

  // A[0] es el máximo elemento, lo pongo al final.
  max = A[0]
  desencolar(A, 0, i) // es la operación de heap sobre arreglo.
  A[i] = max
- ¿Cuál es el invariante?
  - $A[i \ldots n-1]$ está ordenado,
  - $A[0 \ldots i-1]$ es un heap
- ¿Cuál es la complejidad? $O(n) + O(n \ \log n) = O(n \ \log n)$

- Hoy vimos:

(26) Lo que pasó y lo que viene

- Hoy vimos:
  - Colas de prioridad.

(26) Lo que pasó y lo que viene

- Hoy vimos:
  - Colas de prioridad.
  - Implementación sobre heaps.

- Hoy vimos:
  - Colas de prioridad.
  - Implementación sobre heaps.
  - Los heaps tienen complejidad $O(\log(n))$ para encolar y para desencolar, y $O(1)$ para ver el elemento más prioritario.

(26) Lo que pasó y lo que viene

- Hoy vimos:
  - Colas de prioridad.
  - Implementación sobre heaps.
  - Los heaps tienen complejidad $O(\log(n))$ para encolar y para desencolar, y $O(1)$ para ver el elemento más prioritario.
  - Dado un arreglo, el algoritmo de Floyd lo convierte en un Heap en $O(n)$ operaciones.

(26) Lo que pasó y lo que viene

- Hoy vimos:
  - Colas de prioridad.
  - Implementación sobre heaps.
  - Los heaps tienen complejidad $O(\log(n))$ para encolar y para desencolar, y $O(1)$ para ver el elemento más prioritario.
  - Dado un arreglo, el algoritmo de Floyd lo convierte en un Heap en $O(n)$ operaciones.
- Lo que viene:

(26) Lo que pasó y lo que viene

- Hoy vimos:
  - Colas de prioridad.
  - Implementación sobre heaps.
  - Los heaps tienen complejidad $O(\log(n))$ para encolar y para desencolar, y $O(1)$ para ver el elemento más prioritario.
  - Dado un arreglo, el algoritmo de Floyd lo convierte en un Heap en $O(n)$ operaciones.
- Lo que viene:
  - Divide and Conquer.

(26) Lo que pasó y lo que viene

- Hoy vimos:
  - Colas de prioridad.
  - Implementación sobre heaps.
  - Los heaps tienen complejidad $O(\log(n))$ para encolar y para desencolar, y $O(1)$ para ver el elemento más prioritario.
  - Dado un arreglo, el algoritmo de Floyd lo convierte en un Heap en $O(n)$ operaciones.
- Lo que viene:
  - Divide and Conquer.
  - Despedida.