Heap binario

[7541/9515] Algoritmos y Programación II

Abril Diaz Miguez

En este apunte, se repasan los conceptos de heap. Qué es, para qué sirve, las operaciones básicas que podemos aplicarle a él. A lo largo del escrito, también se analizan las ventajas de esta estructura de datos.
Pero vayamos de a poquito, así entendemos bien los conceptos.

Primero que nada, ¿qué es un heap?

Un heap binario es un tipo de estructura de dato, que cumple ciertas características. Nos sirve para almacenar datos, cuando esas características nos convienen.

Ya conozco varias estructuras de datos, ¿para qué quiero otra?

Buena pregunta. La realidad es que no existe una estructura de datos mejor que las demás. Todo depende de qué queremos almacenar y qué queremos hacer con esos datos. Si queremos tener una estructura que cumpla FIFO o FILO, una cola y una pila son las estructuras convenientes, respectivamente. Si nuestro objetivo es agregar y eliminar mucho de la estructura, un abb es una muy buena opción. Si necesitamos realizar muchas búsquedas de datos, un hash es ideal.

De la misma manera, el heap tiene su propósito de existencia. Las dos principales aplicaciones del heap, de las cuales hablaremos más adelante en este apunte, son el *heap_sort* y las colas con prioridad.

Con esta introducción hecha, hilemos más fino.

Ahora la respuesta teórica, ¿qué es un heap?

Un heap binario es un árbol binario con algunas características copadas. Recordemos que un árbol binario es uno en donde los datos están almacenados de manera tal que cada nodo puede tener 0, 1 ó 2 hijos. No se aclara nada sobre la relación de los nodos.

Un heap binario sí presenta cierta relación entre elementos. No hay un criterio de orden total, no hay relación entre los hermanos, los sobrinos y tíos, o cualquier pariente similar. Lo que sí hay es un orden parcial, una relación de rama. ¿A qué nos referimos con esto? Si miramos solamente una rama, existe un orden, que depende del tipo de heap que se tenga.

Primera característica

Supongamos que tenemos un heap maximal. El orden que se respeta es que cuanto más cercano esté nuestro elemento de la rama al nodo raíz, más grande será en comparación a los elementos que estén por debajo de él… No se entendió nada, ¿no? Genial, veamos un gráfico.

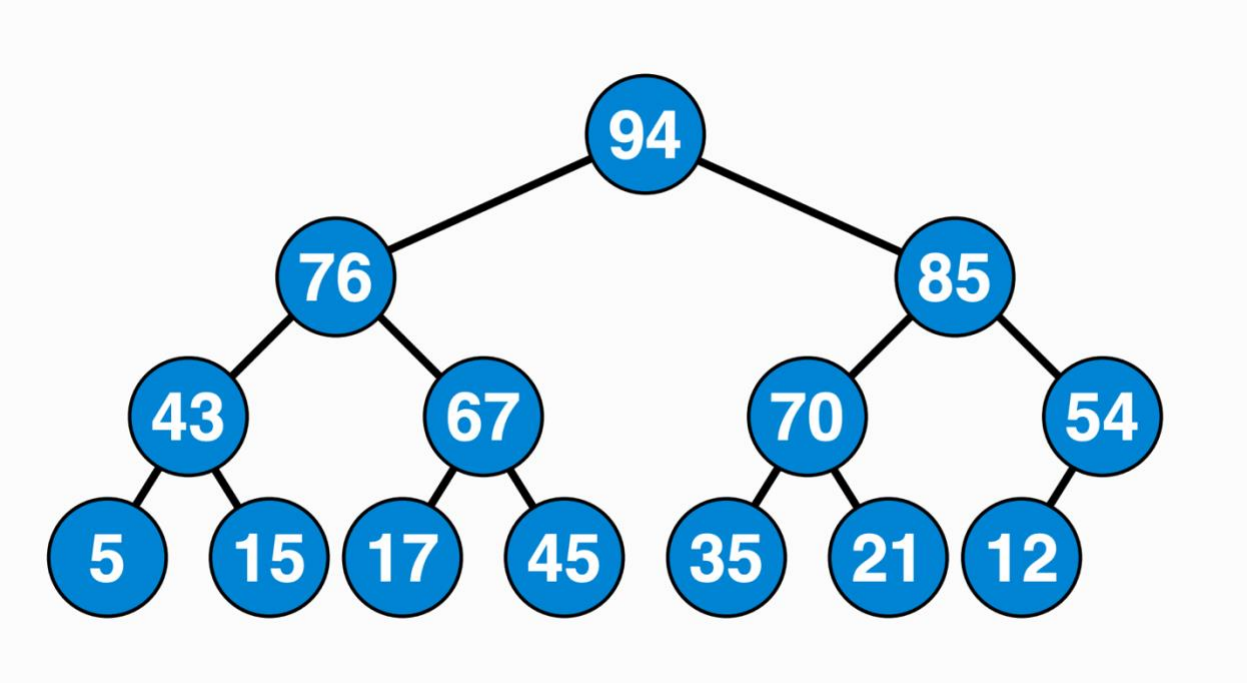


Figura 0

Acá tenemos nuestro heap. Marquemos algunas ramas y encontremos el máximo de la misma.

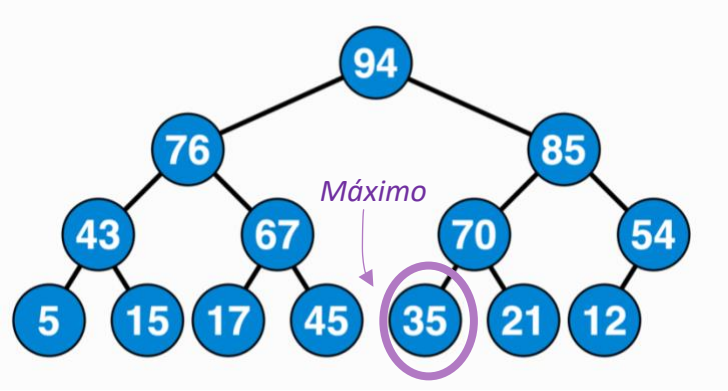


Figura 1

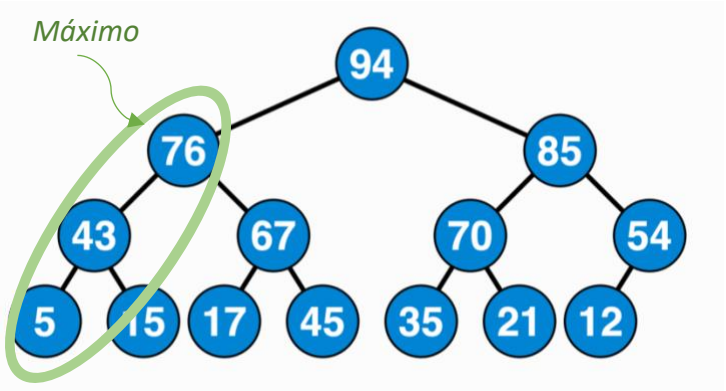


Figura 2

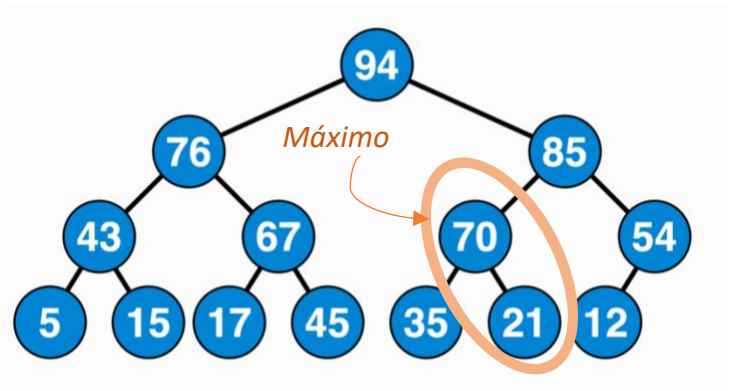


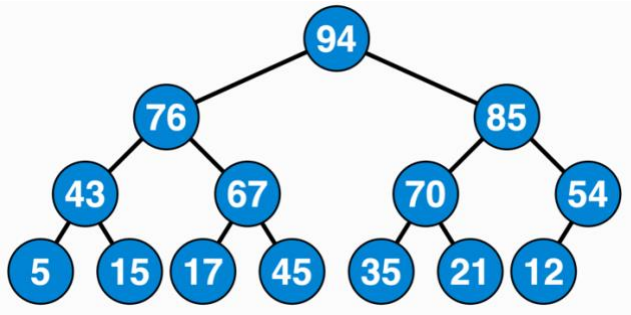
Figura 3

Tanto en la *figura 1* como en la *2* y *3* se marcaron con color diferentes ramas del heap principal. En la *figura 1*, la rama está compuesta por el elemento 35, así que ésa es la raíz. En la *figura 2*, la rama está compuesta por los elementos 76, 43 y 15, y la raíz es el 76. En la *figura 3*, la rama está compuesta por los elementos 70 y 21, y la raíz es el 70.

Vemos que la raíz de cada rama marcada es el elemento más grande de esa rama. Y si mirásemos cada rama que tiene al 94 como raíz, veríamos que el 94 es siempre el número mayor. Ésta es una característica de los **heaps maximales**: **el nodo con el mayor elemento se encuentra en la raíz de la rama actual.** Si tuviésemos un heap minimal, el nodo con el menor elemento se encontraría en la raíz de la rama actual. Si ahora tomamos a la rama actual como la que involucra a la raíz del árbol, podemos concluir que el mayor (o menor) elemento del heap se encuentra en la raíz de todo el árbol. Es decir, *la raíz es el mayor (o menor) elemento presente en el heap, como una de las condiciones de validez del heap*. Y tengo acceso directo a ese elemento, lo que será de gran ventaja.

Segunda característica

Otra característica de los heaps es que se tratan de árboles casi completos. ¿A qué nos referimos con esto? Comparemos un abb típico y un heap típico.



Heap típico

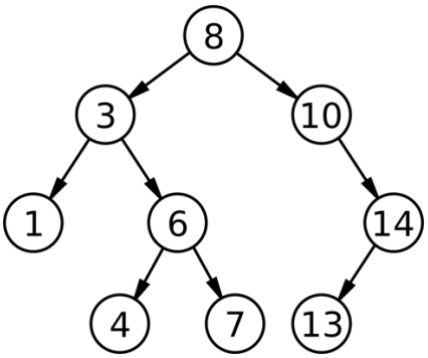
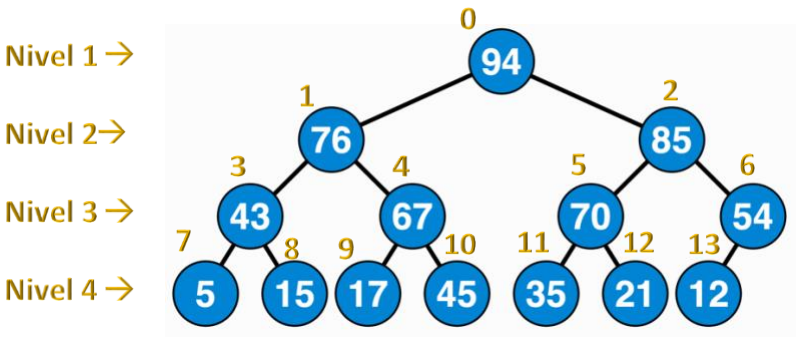


ABB típico

Examinemos el abb típico primero. Vemos que la raíz, el 8, tiene sus dos hijos, al igual que el elemento 3, 10 y 6. Notamos que los elementos 1, 4, 7 y 13 son hojas y que el elemento 14 tiene un hijo. En contraste, las únicas hojas que el heap típico presenta son las que se encuentran en la fila 4, en el último nivel. Agreguemos ahora la posición de cada elemento, desde el punto de vista de filas.



Heap típico con posiciones

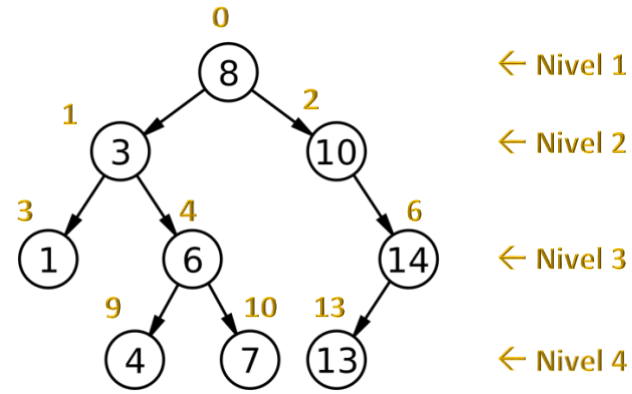


ABB típico con posiciones

Notamos que en el abb típico el elemento 1 (en la posición 3) no tiene hijos, pero el abb sigue siendo completamente válido. Un abb puede contener hojas en niveles intermedios, y no presenta ningún problema. En contraste, un heap no puede poseer hojas en niveles intermedios; por eso vemos que el heap típico está completo hasta el último nivel. A esto nos referimos cuando establecemos que un heap es un árbol binario casi completo. Siempre vamos a completar los niveles de manera ordenadas, no vamos a tener agujeros o espacios vacíos en un nivel intermedio. Es decir, no puedo empezar un nuevo nivel si hay espacios vacíos en el actual.

A primera vista nos puede parecer como una clara desventaja del heap. El abb aparenta ser más flexible que un heap, menos riguroso y más sencillo de usar. Pero no olvidemos que la flexibilidad siempre viene a un costo, y que la rigurosidad muchas veces viene a beneficio del usuario. En este caso, el costo es la necesidad de usar memoria dinámica para implementar el abb. Porque claro, la memoria dinámica no es la única forma de implementar estructuras de datos, ¿no? Recordemos que cuando aprendimos sobre listas, se mencionó que se pueden implementar con vectores; pero no se dijo lo mismo de abbs. ¿Por qué?

La realidad es que no es conveniente implementar un abb con un vector justamente porque no es una estructura completa. Un abb típico puede presentar varios agujeros en su estructura, y cada agujero equivaldría a un espacio vacío en el vector. Si pensamos en un árbol grande, que almacena structs complejos, deberíamos solicitar mucha memoria para almacenarlo correctamente, pero probablemente terminemos usando solamente una fracción de la misma. Es demasiada memoria perdida.

En contraste, un heap es una estructura completa. Esto es, no posee agujeros en su estructura, y almacenarlo en un vector es increíblemente eficiente. Es más, si conocemos una buena cota de cuántos elementos va a haber en nuestro heap, ni siquiera necesitamos hacer uso de la memoria dinámica, podemos inicializar un vector en el stack. Usar el stack por encima del heap involucra menos riesgo, y utilizar vectores por sobre punteros hace que el movernos a través de la estructura sea más fácil. Se detalla cómo utilizar vectores para construir heaps en la sección correspondiente.

Tercera característica

Pregunta: ¿se puede eliminar cualquier elemento de un heap? ¿Y buscar un elemento cualquiera? ¿O leer su contenido? La respuesta es un no cauteloso, porque ¿para qué querrías eliminar cualquier elemento de un heap?

Un heap es una estructura que nos da mucha información de la raíz, pero poca del resto de los nodos. Se utiliza el heap cuando queremos interactuar siempre con el más grande (o más chico) de los elementos, así que eliminar, buscar o leer un elemento en el medio del heap no tiene mucho sentido. Si querés interactuar con todos los nodos del árbol por igual, es más conveniente usar un abb, por ejemplo.

Llegamos así a nuestra tercera característica, la cual es que ***solamente podemos acceder al elemento raíz del heap***. Repito, solamente podemos leer, buscar y eliminar el elemento raíz del heap. Esta característica es de suma importancia, ya que es la condición que nos permite tener la operación búsqueda con complejidad algorítmica O(1). ¿Por qué? Porque tenemos acceso directo a la raíz, en todo momento. Por lo tanto, leer su contenido es una operación con complejidad O(1), una gran ventaja en contraste con la búsqueda de un abb, que tiene complejidad O(log n) y la búsqueda de una lista, que tiene complejidad O(n).

Repaso rápido

Repasemos las propiedades que un árbol binario debe cumplir para ser un heap binario

- *Debe estar ordenado por ramas*. Esto quiere decir que, si seleccionamos una rama cualquiera del árbol, hay un orden descendiente (si se trata de un heap minimal) o ascendiente (si se trata de un heap maximal).
    - Como consecuencia del punto anterior, *la raíz del árbol es el menor o mayor elemento de todo el heap,* si se trata de un minimal o de un maximal, respectivamente.
- *Todos los niveles del heap, exceptuando el último, están completos.*
- *Sólo se puede leer, buscar y borrar la raíz del heap.*

¿Para qué quiero un heap?

Ya establecimos (varias veces) que el máximo de todo el heap está en la cabeza del heap, en la raíz. Sabemos que en un árbol tenemos un puntero a la raíz, por lo que buscar la raíz es una operación de complejidad O(1). Es lógico concluir que si buscamos tener el elemento más grande a mano (o más chico, dependiendo del orden que se eligió), un heap nos viene bárbaro.

Un ejemplo de utilización del heap es el método de ordenamiento heap sort. Este método recibe el vector a ordenar y, al finalizar la ejecución de la función, ese mismo vector contiene los elementos ordenados. Este algoritmo, el cual no se detallará en este apunte, realiza la conversión del vector a un heap (de ser necesario) y el ordenamiento propiamente dicho. La función presenta una complejidad algorítmica de O(n log n), mas no es estable.

¿Qué quiere decir que no es estable? Un método de ordenamiento es estable cuando los elementos repetidos mantienen el orden relativo entre sí incluso luego del ordenamiento. ¿Y qué queremos decir con esto? Supongamos que tenemos el listado de padrones, que ordenamos de menor a mayor padrón. Si más tarde decidimos reordenar ese listado, ahora en función al apellido del alumno, seguramente encontremos que varios alumnos tienen el mismo apellido. En ese caso, un método estable mantendría el previo orden de los elementos, resultando en que los alumnos con igual apellido queden ordenados de menor a mayor padrón. Si el método no es estable, no hay un orden establecido para los alumnos con igual apellido.

Otro ejemplo de utilización de heap es implementar una cola con prioridad. Sabemos que una cola respeta el principio FIFO o first-int first-out, que quiere decir que el primer elemento en ingresar a la cola será el primero en salir; el segundo en ingresar será el segundo en salir. Al igual que en la cola del supermercado, donde el primero que ingresa a la cola es el primero que es atendido por el cajero. Sin embargo, a veces necesitamos poder priorizar ciertos elementos en la cola. Siguiendo el ejemplo del supermercado, si llega una mujer embarazada o un señor mayor, lo correcto es dejarlos pasar primero. Pero se estaría dejando de cumplir el principio FIFO, ¿no? Una implementación típica de una cola no da soporte a ese tipo de comportamiento… ¿cómo lo resolvemos?

Introduciendo la estructura de heap. Ahora, en vez de mantener a los elementos en una cola, los ingresamos a un heap, con un cierto valor de prioridad. Podríamos, por ejemplo, ingresar a las personas sin prioridad en la cola a partir de cierto valor en adelante. El que llega primero recibe el valor 100, el que llega segundo, el 101, y así. Si almacenamos estos valores en un heap minimal, la raíz siempre apuntará a la persona que debe ser atendida en ese momento por el cajero. Cuando llegue una persona que debe pasar primero, le asignamos un valor más bajo de 100 de prioridad, por ejemplo, 20. El heap minimal, como debe mantener siempre al elemento con menor valor en la raíz, se reorganizará para que el siguiente elemento que extraiga del mismo sea el 20, sin importar que haya llegado último.

Ya vimos qué es un heap y para qué sirve. Pasemos entonces a revisar las principales operaciones que a él se le aplican.

Operaciones del heap

Insertar

Desordenemos un poco nuestro heap. Quiero insertar un nuevo elemento en él. Por supuesto que lo vamos a insertar en donde tenemos el primer espacio libre que, sí, será en el último nivel. ¿Por qué? Porque dijimos que el heap era un árbol casi completo, no puedo tener agujeros en el medio de la estructura.

Insertamos el elemento 90.

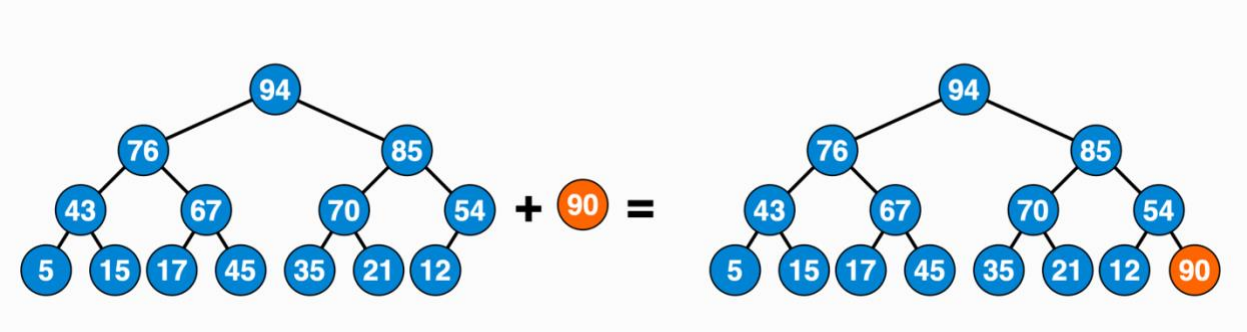


Figura 4

Pero guarda, esto ya no es un heap maximal. Fijate que esta rama, la que se muestra en la *figura 5*

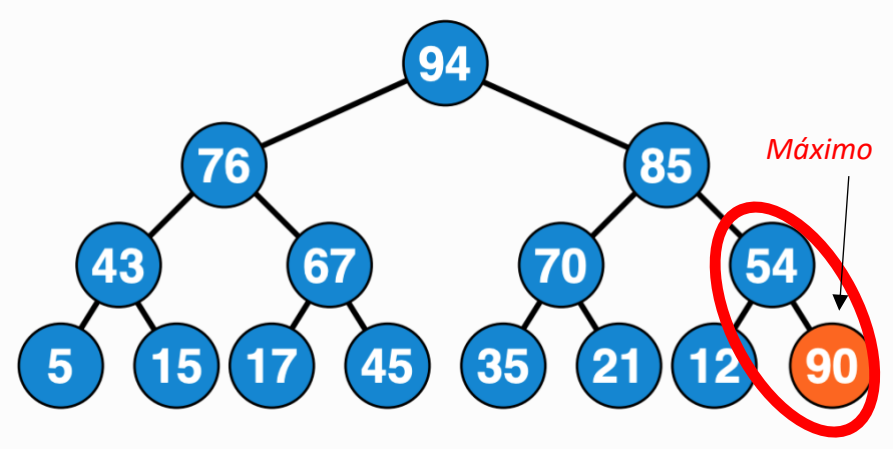


Figura 5

ya no está ordenada como debería; la raíz de esa rama no es el mayor de esa rama. Por lo tanto, no se cumple la condición de heap maximal y ya no tenemos un heap válido.

Para ordenarlo y tener un heap válido, aplicamos la operación *sift up*. Esta operación involucra al nodo actual y al padre. El nodo actual le pregunta al padre "¿Soy mayor que vos?", si la respuesta es sí, se intercambian (el padre pasa a ser el hijo y el hijo pasa a ser el padre); si la respuesta es no, es porque el padre es mayor al nodo actual, así que el nodo actual está en la posición correcta.

De nuevo, entendamos bien con gráficos.

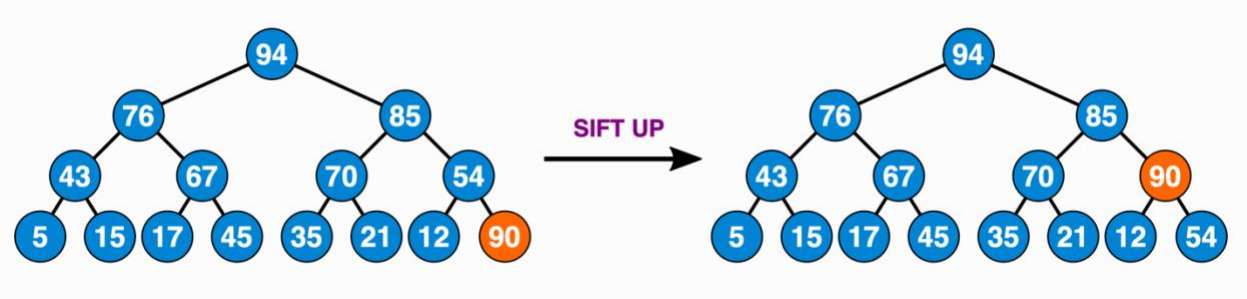


Figura 6

En la *figura 6*, comparamos el 90 con el 54. Como el 90 es mayor al 54, el 90 debería estar más arriba en la rama que el 54. Así que los intercambiamos.

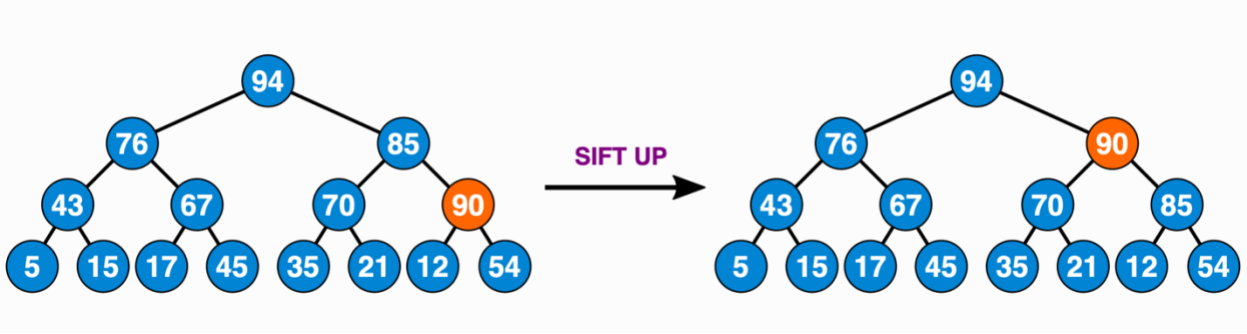


Figura 7

En la *figura 7*, comparamos el 90 con el 85. Como el 90 es mayor al 85, el 90 debería estar más arriba en la rama que el 85. Así que los intercambiamos.

En la tercera iteración, comparamos 90 con 94. El 94 es mayor al 90, así que el 94 se mantiene como la raíz, y encontramos el lugar adecuado del 90.

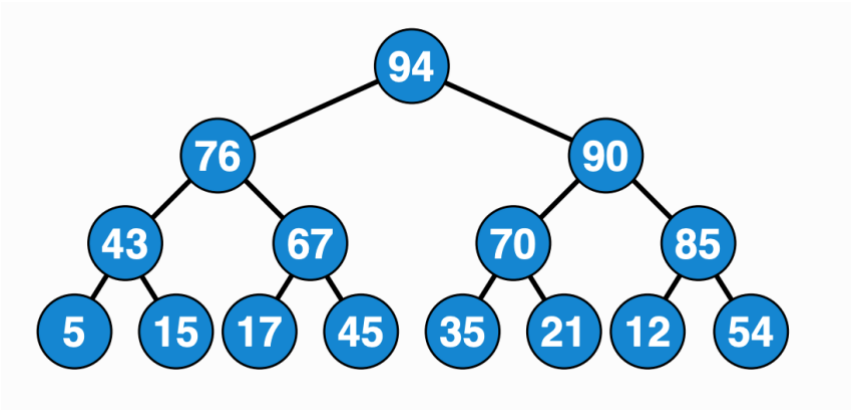


Figura 8

Ya sabemos insertar, ¡genial!

Eliminar el máximo (de un maximal) o el mínimo (de un minimal)

Sigamos rompiendo nuestro heap (y arreglándolo después, por supuesto). Eliminar el elemento máximo de un heap maximal. Lo primero que hacemos es, por supuesto, borrar ese elemento. ¡Pero nos queda un espacio vacío! Se deja de cumplir una de las condiciones de heap, la que establece que el heap es una estructura casi completa. Entonces, tenemos que rellenar ese agujero con algo.

Lo más sencillo de encajar ahí es una hoja, ¿no? Porque no tiene hijos y no aparecería un nuevo agujero a rellenar. Para mantener la condición de que el heap es una estructura casi completa (y no generar nuevos agujeros), reemplazamos la raíz con el último elemento del heap.

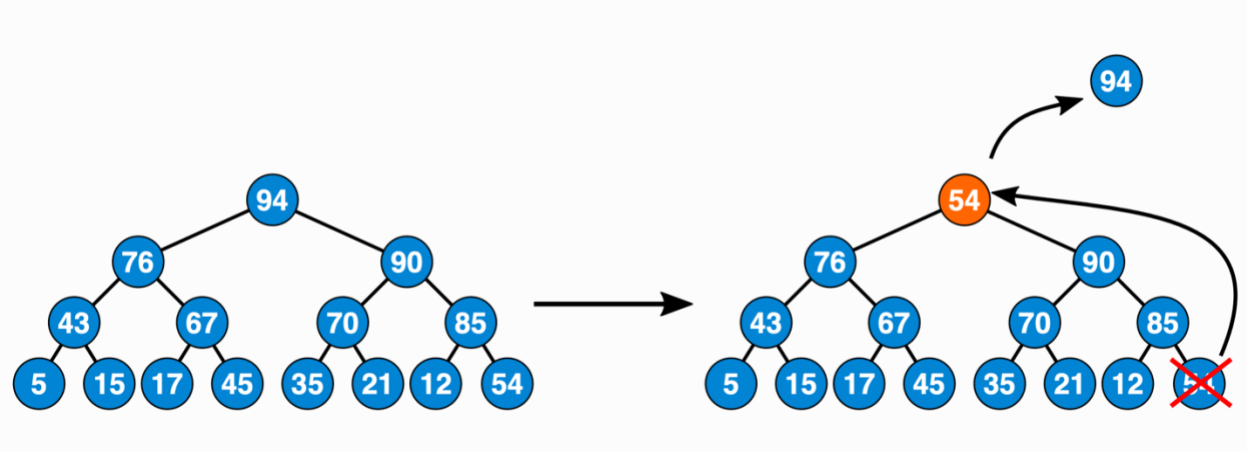


Figura 9

Ya no tenemos agujeros en el heap, bien. El problema es que nuestro "heap" se volvió a romper, porque ahora el elemento en la raíz no es el mayor de todo el heap. ¿Cómo sé que no es el mayor del heap? Porque antes era una hoja de una rama, por lo que el padre de esa hoja era mayor a la hoja. Tengo que colocar a esa nueva raíz en su lugar correcto.

Para lograrlo, usaremos la función *sift down*. Esta operación compara al nodo actual con todos sus hijos directos, e intercambia al actual con el hijo mayor. Si no hay hijo mayor, el nodo actual es el mayor de los tres, y encontramos el lugar correcto del nodo actual.

Como siempre, veamos el ejemplo con gráficos.

Figura 10

En la *figura 10*, comparo el 54 con los valores 76 y 90. El mayor número de los tres es el 90, que es uno de los hijos, así que intercambiamos el 54 (el nodo actual) con el 90 (el hijo mayor).

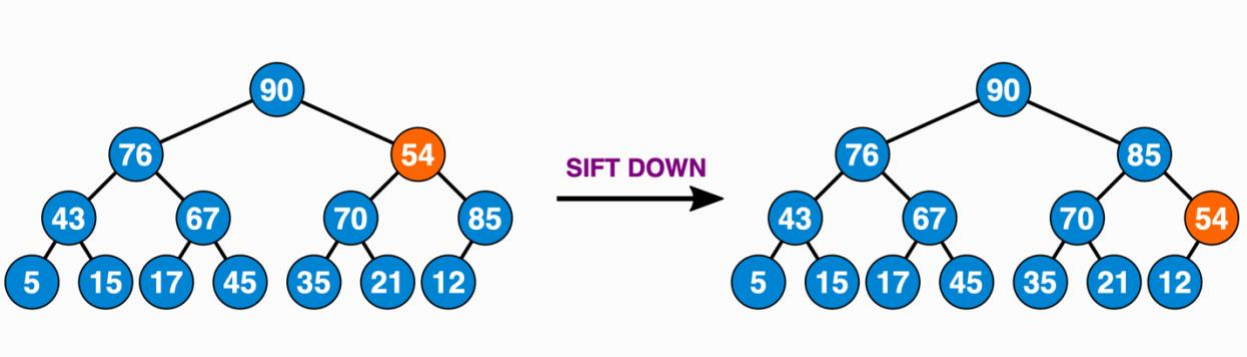


Figura 11

En la *figura 11*, comparo el 54 con los valores 70 y 85. El mayor número de los tres es el 85, que es uno de los hijos, así que intercambiamos el 54 (el nodo actual) con el 85 (el hijo mayor).

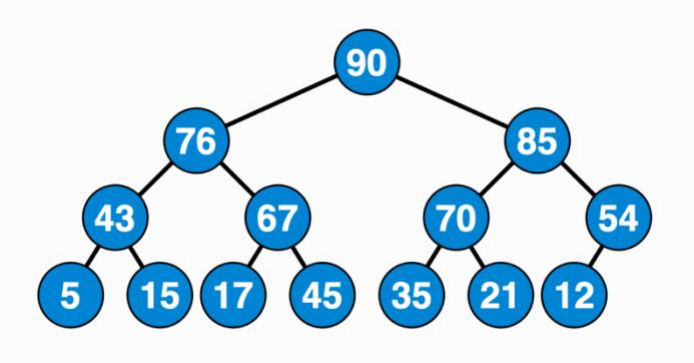


Figura 12

En la tercera iteración, comparamos el 54 con el 12. Como el nodo actual, 54, es mayor a los hijos (12 y nada), encontramos el lugar adecuado para el nodo 54.

Repetime, ¿por qué un heap y no un abb?

Los heaps se usan cuando necesitamos tener a mano el elemento más grande o más chico de un conjunto. Las operaciones de inserción y eliminación tienen ambas una complejidad O(log n), y obtener el mayor o menor tienen ambas una complejidad O(1). Por lo que, si necesitás interactuar siempre con el mayor o menor, esta estructura es mucho mejor que una lista.

¿Y por qué usaríamos un heap y no un abb? Si ambas tienen la misma complejidad de inserción y eliminación…

Buena pregunta. La clave está en que no tienen la misma complejidad de búsqueda. En un heap, siempre se busca el mayor (o el menor), que tiene complejidad O(1). En un abb, la cota de la complejidad de búsqueda es siempre O(log n).

No te olvides que elegimos la estructura de datos basándonos en qué queremos hacer nosotros con ella. Si vamos a insertar, eliminar y buscar repetidamente, hay estructuras que están hechas para eso, como un abb o una tabla de hash. Si buscamos simular una pila o cola, es preferible usar esos tdas. Si queremos una cola con prioridad, es conveniente un heap. No hay una estructura mejor que otra, todo depende de qué queramos hacer con los datos que en ella guardemos.

Construcción

Hace varias páginas, comparamos las estructuras del abb y del heap, y llegamos a la conclusión de que almacenar un heap en un vector es altamente eficiente y recomendable. Veamos cómo hacerlo.

En la *figura 13*, vemos un heap minimal en la estructura ya conocida, y un vector que lo almacena. Para movernos a lo largo del vector como si fuese un heap, debemos conocer la relación entre el nodo actual y sus dos hijos, que son las mencionadas en la figura.

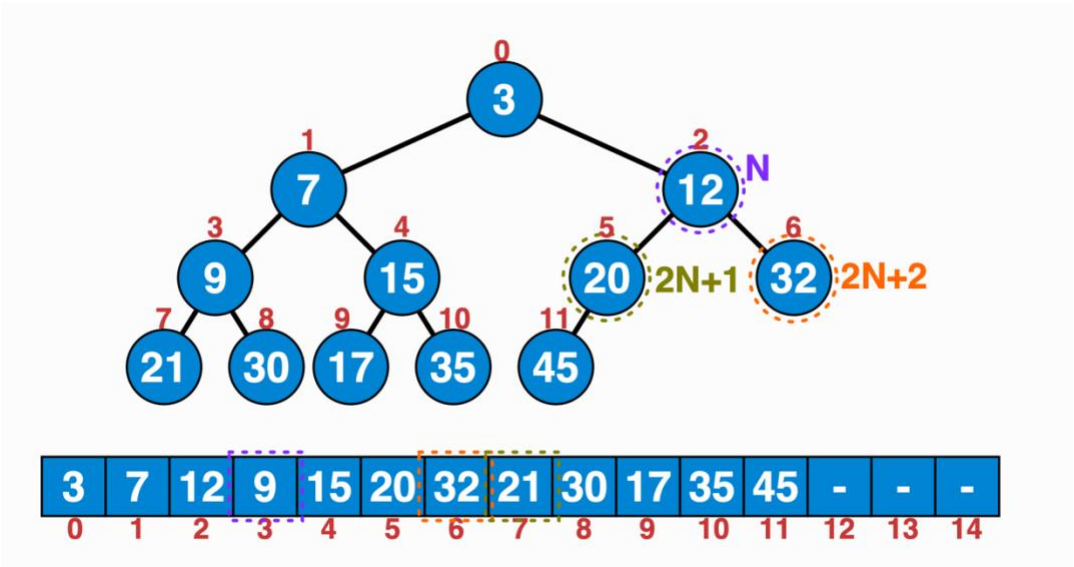


Figura 13

Si el nodo actual está en la posición n, empezando desde 0, el hijo izquierdo está en la posición $2n + 1$ y el hijo derecho, en la posición $2n + 2$.

Para construir un heap hay dos opciones. O creamos uno nuevo y vamos insertando correctamente los elementos, u ordenamos el mismo vector con el método *heapify*. Esta función recibe un vector que no tiene agujeros, como el que se muestra en la *figura 13*, y lo reordena para que cumpla las condiciones de un heap, ya sea minimal o maximal.

Dado el vector de la *figura 13*, que podemos ver que no es un heap ni minimal ni maximal, apliquemos el algoritmo heapify.

El algoritmo empieza averiguando cuántos elementos podemos tener en nuestro heap. ¿En un heap de 4 niveles, cuántos elementos puedo tener? Miremos por filas. En la primera tengo 1 elemento, que se puede escribir como 2^0; en la segunda fila tengo 2 elementos, que es igual a 2^1; en la tercera fila tengo 2^2 elementos y en la cuarta fila puedo tener hasta 2^3 elementos. Si sumamos, tenemos $2^0 + 2^1 + 2^2 + 2^3 = 15 = 2^4 - 1$. Como regla general, se dice que en un heap podemos tener hasta $2^n - 1$ elementos, si n es la cantidad de niveles del heap.

Con eso dicho, sabemos que nuestro heap tiene 4 niveles y, como máximo, 15 elementos. A cada elemento se le va a intentar aplicar la función *sift down*, para generar un heap minimal. Ahora, ¿tiene sentido aplicarle la función *sift down* a los de la última fila? Si son hojas, ¡no tienen hijos! La función *sift down* no va a hacer nada, no hay dos nodos a intercambiar. Así que tenemos que empezar desde la anteúltima fila, que seguro contiene aunque sea un nodo con hijos.

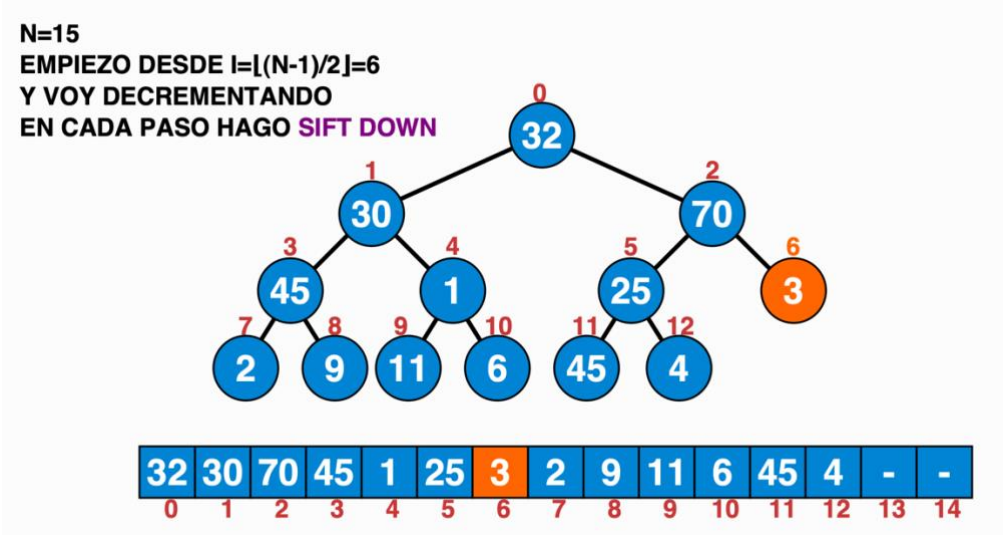


Figura 14

Empezamos desde el último elemento de la anteúltima fila. En este caso, es el elemento 3, que está en la posición 6. Aplicamos *sift down*, y no pasa nada porque el elemento 3 no tiene hijos, así que me muevo hacia el 25 (que está en la posición 5).

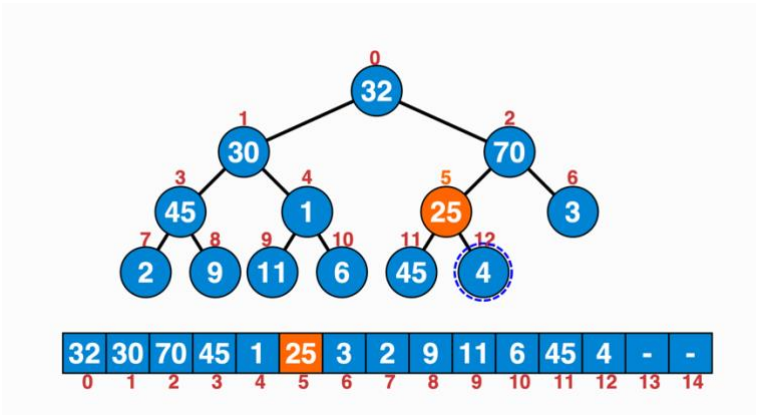


Figura 15a

En la *figura 15a* estamos parados en la posición 5, con el valor 25. Comparamos al 25 con sus hijos para buscar el menor. Descubrimos que el 25 tiene como hijo al elemento 4, en la posición

$2 * n + 1$, con $n = 5$, que es la posición del nodo actual. Así que el elemento 4 está en la posición $2 * 5 + 1 = 12$. Intercambiamos el 25 por el 4, como se muestra en la *figura 15b.*

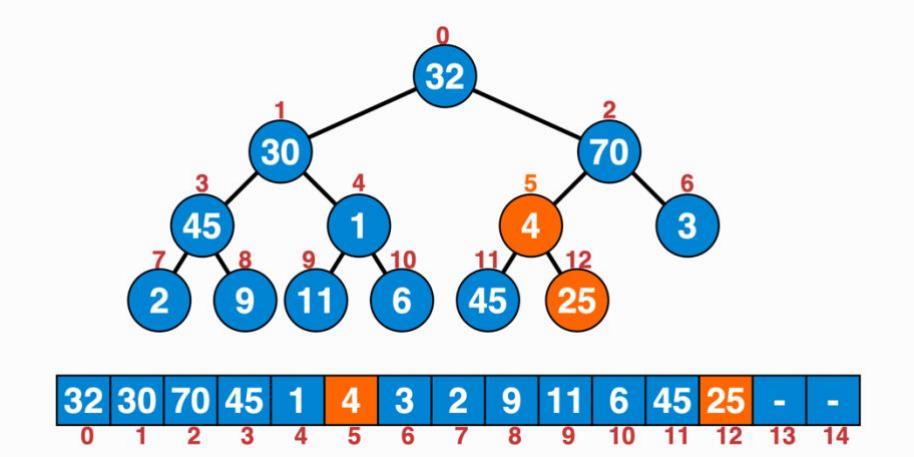


Figura 15b

Nos movemos a la siguiente iteración en la *figura 16*, parándonos en el elemento 1, que está en la posición 4.

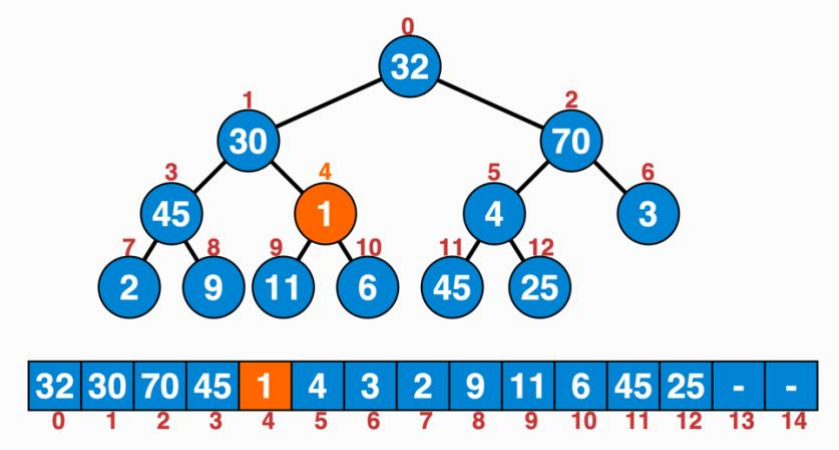


Figura 16

Otra vez, comparamos el nodo actual con sus dos hijos y verificamos que el nodo actual es el menor de los tres, así que no intercambiamos ningún elemento.

En la *figura 17a* nos movemos al siguiente elemento, el 45, que está en la posición 3.

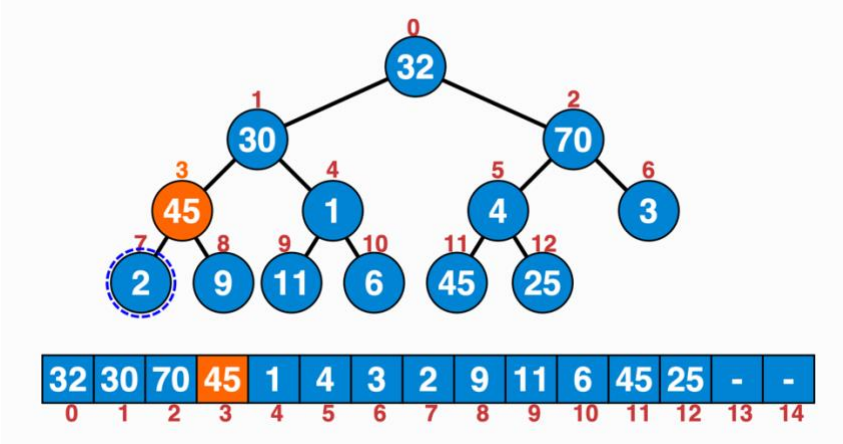


Figura 17a

Comparamos el nodo actual con sus dos hijos y verificamos que uno de los hijos es el menor de los tres, así que intercambiamos ese hijo menor por el nodo actual, como se muestra en la *figura 17b.*

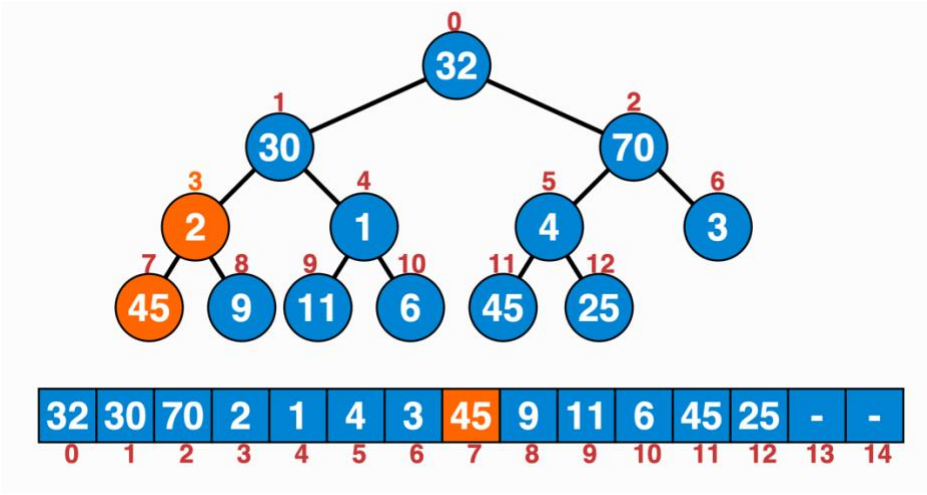


Figura 17b

Seguimos a la siguiente iteración en la *figura 18a*. Nos paramos en el nodo 70, en la posición 2, y verificamos que sus dos hijos son menores a él. Tomamos el menor de los dos hijos y lo intercambiamos por el nodo actual en la *figura 18b.*

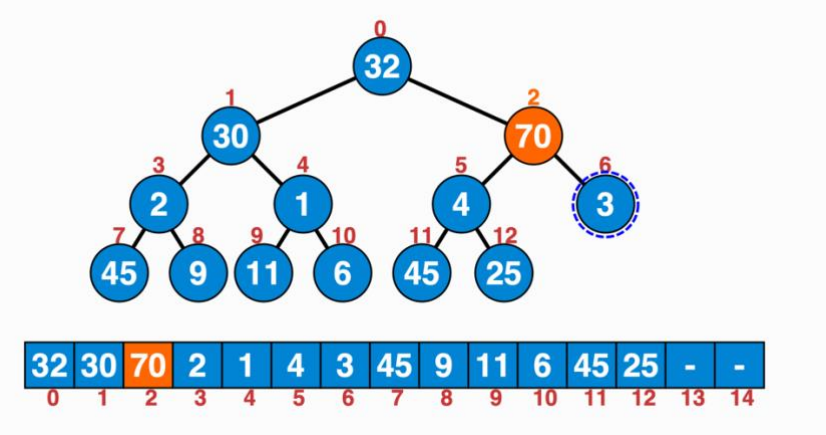


Figura 18a

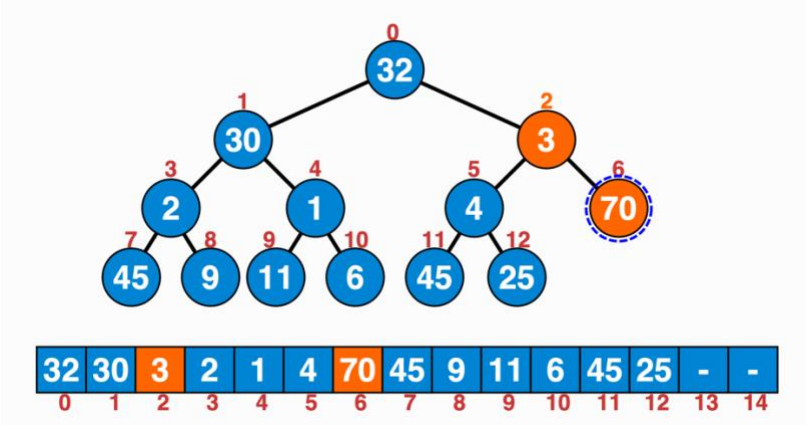


Figura 18b

Pasamos a la siguiente iteración en la *figura 19a*. Nos movemos al elemento 30, que está en la posición 1.

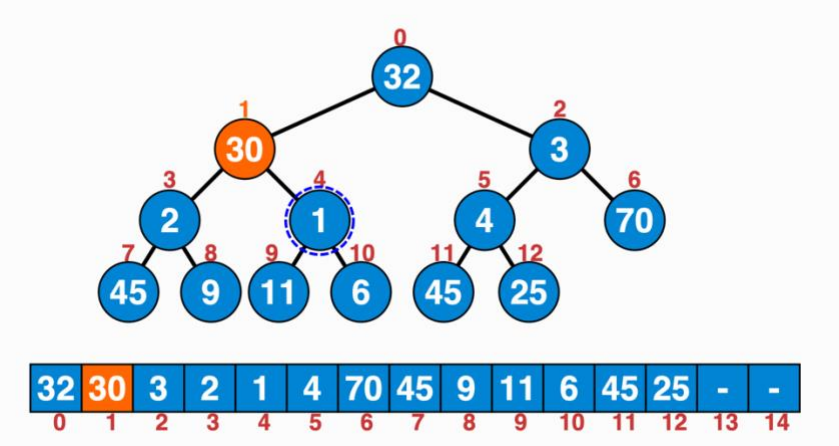


Figura 19a

Vemos que los dos hijos son más chicos que el nodo actual, así que tomamos el menor y lo intercambiamos con el actual en la *figura 19b.*

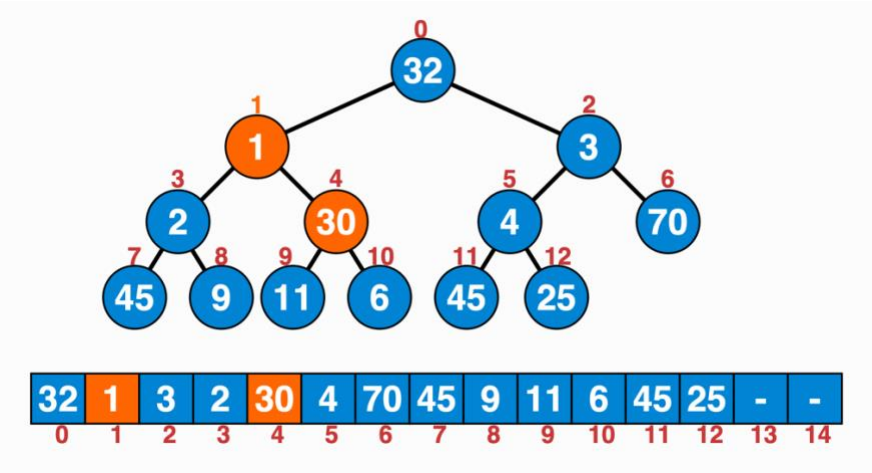


Figura 19b

Pero notamos que al intercambiar el 1 con el 30, el 30 pasó a ser padre del 11 y del 6. Ambos valores son menores a 30, así que no se cumple la propiedad de heap minimal. Tenemos que reordenar esa parte. Así que ahora nos paramos en 30 y vemos sus dos hijos. Elegimos el menor de ambos e intercambiamos, como se ve en la *figura 18c* y en la *figura 18d.*

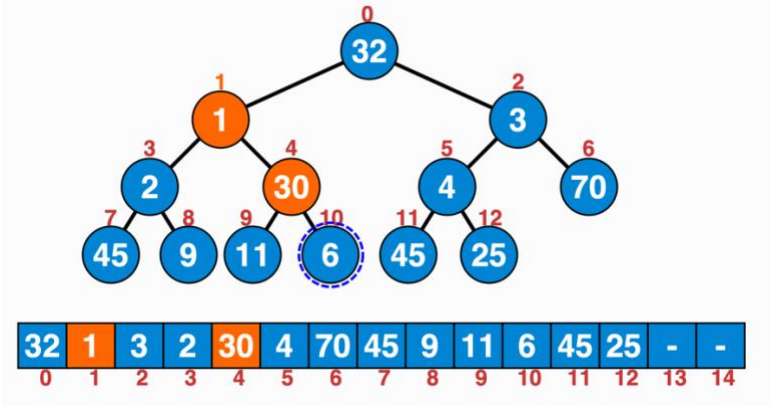


Figura 19c

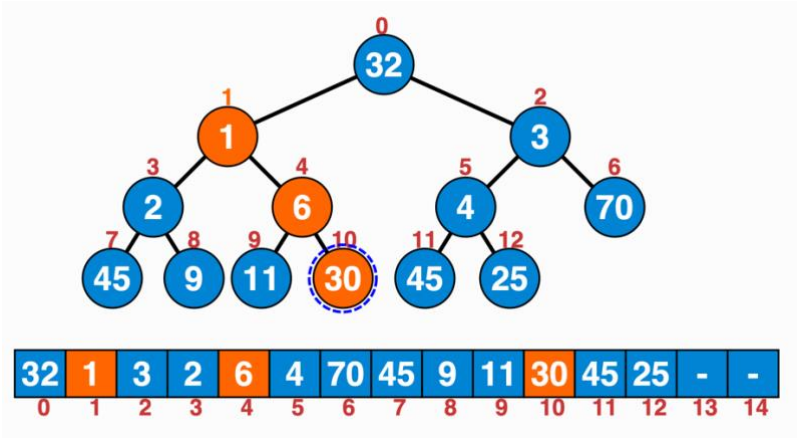


Figura 19d

Y así, obtenemos el elemento 1 y sus hijos ordenados.

Nos falta la raíz, que ya podrás ver que va a ser un proceso similar al de la *figura 19*.

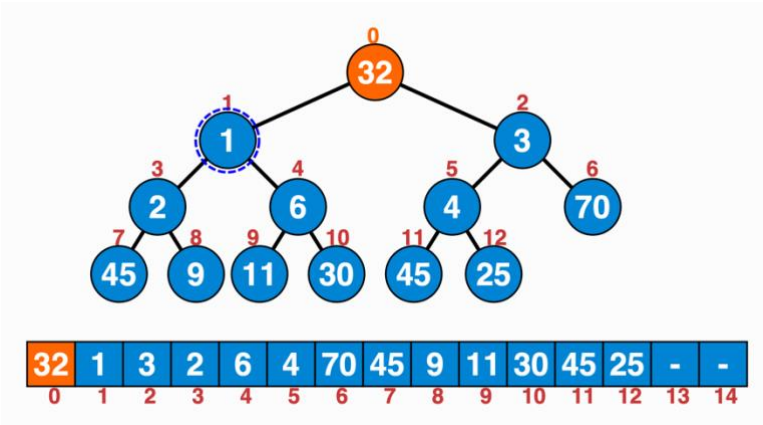


Figura 20a

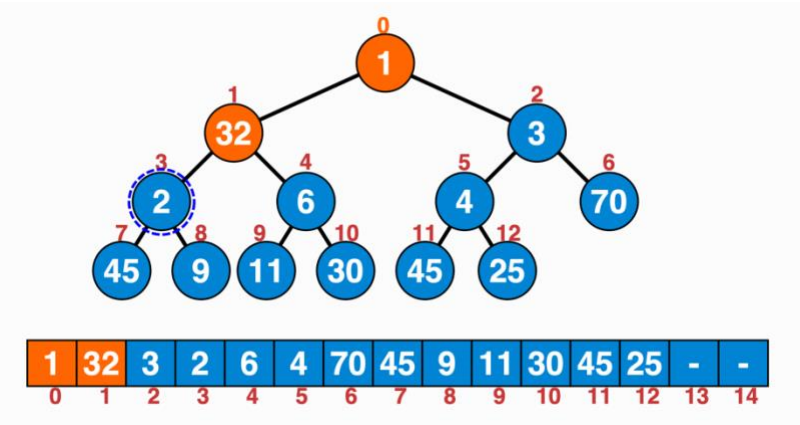


Figura 20b

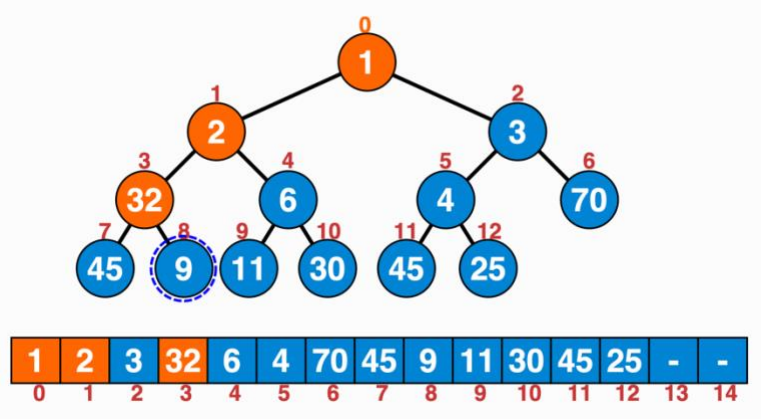


Figura 20c

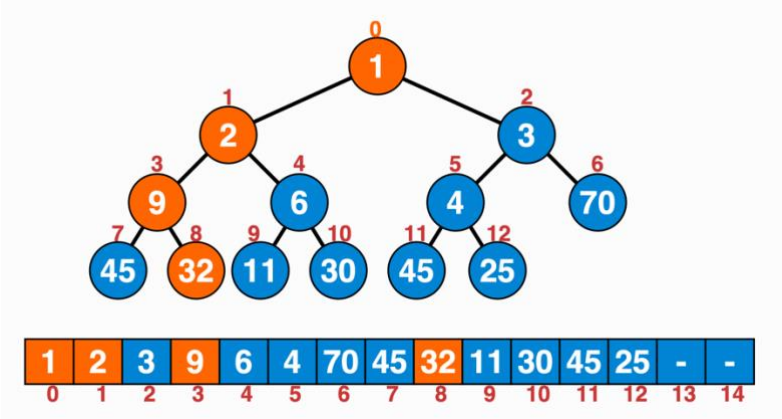


Figura 20d

Finalmente, obtenemos un heap que cumple las condiciones.

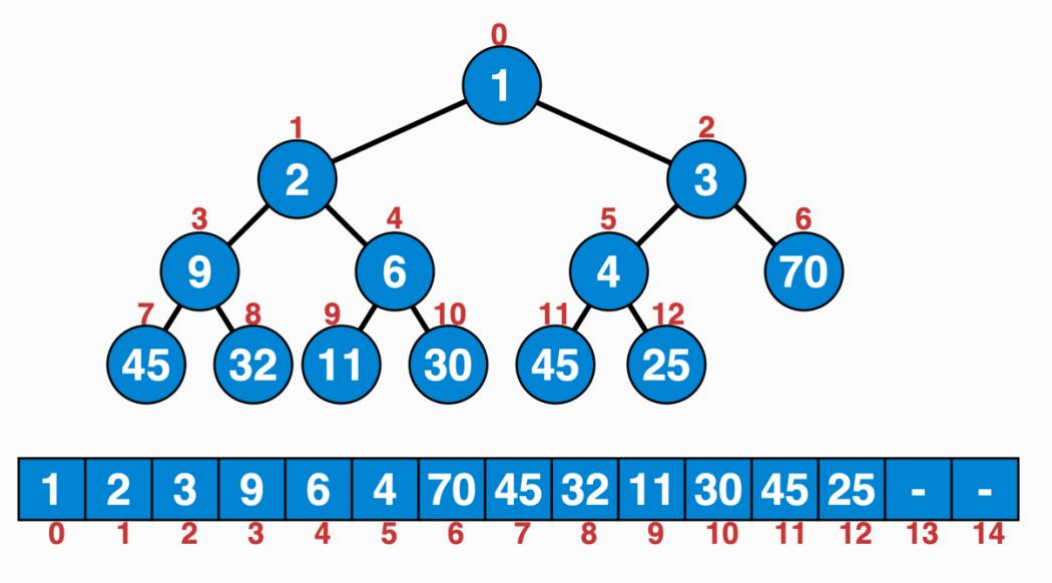


Figura 21

Eso sí, el heap está ordenado, pero si mirás el vector, notarás que al leerlo como un array, no hay un orden establecido. Al ordenarlo gráficamente como un heap, notamos que cumple las condiciones de heap minimal, así que es válido, aunque a primera vista, el vector no parezca cumplirlo.

¿Y la complejidad de *heapify*? Debemos analizar el peor caso siempre. El *sift down* lo tengo que aplicar a todos los nodos menos los de la última fila, así que si tengo n nodos, debo aplicar *sift*

down a n/2 nodos. Como *sift down* tiene complejidad O(log n), la complejidad de *heapify* es O(n/2 log(n)), que nosotros tomaremos como O(n log n).

Ultra mega resumen

Ya vimos qué, cómo y para qué. Ahora, terminemos de cerrar la idea con un buen resumen.

- Un heap es una estructura para almacenar datos. Se trata de un árbol binario que cumple que:
    - *Debe estar ordenado por ramas*. Esto quiere decir que si seleccionamos una rama cualquiera del árbol, hay un orden descendiente, si se trata de un heap minimal, o ascendiente, si se trata de un heap maximal.
        - Como consecuencia del punto anterior, *la raíz del árbol es el menor o mayor elemento de todo el heap,* si se trata de un minimal o de un maximal, respectivamente.
    - *Todos los niveles del heap, exceptuando el último, están completos*.
    - *Sólo se puede leer, buscar y borrar la raíz del heap.*
- Para insertar un elemento al heap, lo agregamos en el primer agujero que se encuentra (que necesariamente está en el último nivel del heap) y con la operación *sift up*, sube de nivel hasta su posición correcta.
- Para eliminar la raíz del heap, quitamos ese elemento y traemos al último elemento del heap, es decir, la hoja que está más a la derecha, a la raíz. Luego, aplicamos la operación *sift down* las veces necesarias para que esa hoja convertida en raíz se posicione en su lugar adecuado.
- Una aplicación del heap es el método de ordenamiento heap_sort, que en promedio tiene complejidad algorítmica O(n log n).
- Otra aplicación del heap es utilizarlo para representar una cola con prioridad.
- Una primera manera de construir un heap es instanciando un nuevo vector e insertar los elementos uno por uno.
- Una segunda manera de construir un heap, ligeramente más óptima (sobre todo porque no requiere de memoria extra) es utilizando el método *heapify*.