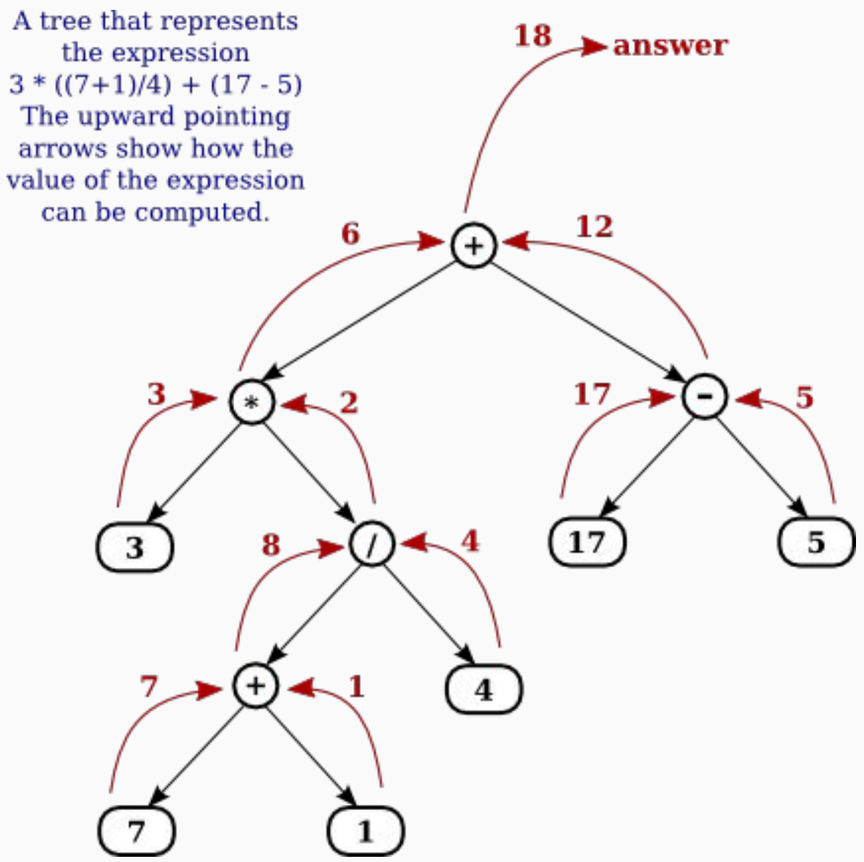
UTEC
UNIVERSIDAD DE INGENIERÍA
Y TECNOLOGÍA

Welcome to Algorithms and Data Structures! - CS2100

Expression trees

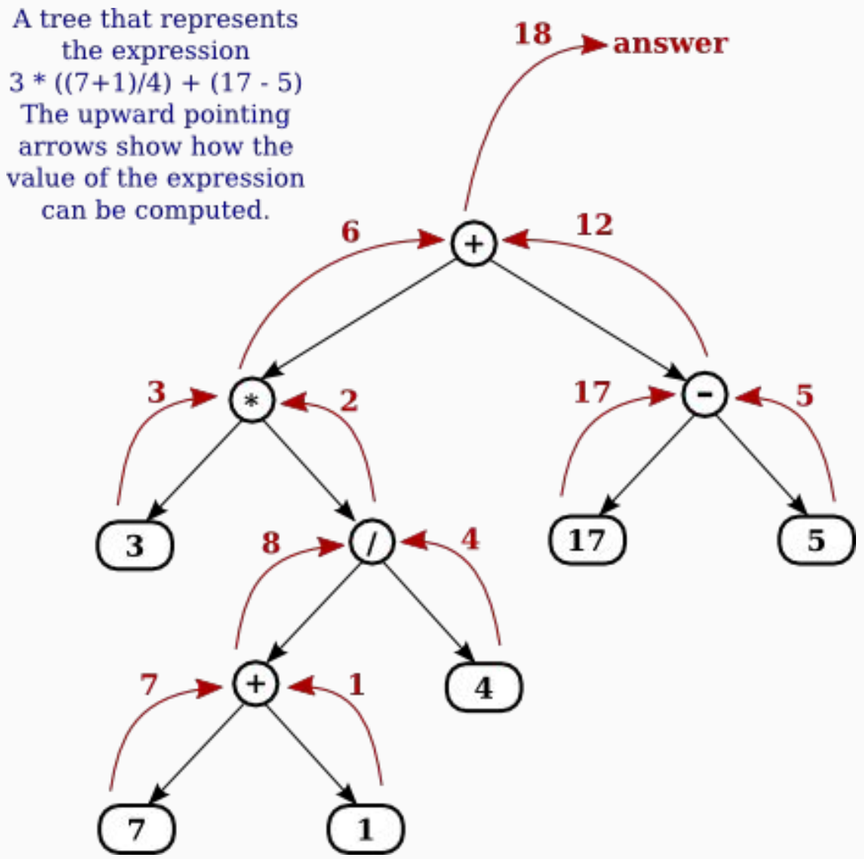
- Es un árbol binario.
- Cada nodo interno corresponde con un operador.
- Cada nodo hoja corresponde con un operando.
- Un recorrido inorder produce una versión infija de la expresión. Lo mismo con preorder, que produce una expresión prefija.

A continuación veremos el algoritmo para transformar de infijo (A + B) a postfijo (A B +)



Expression trees

1. Se recorre la expresión de izquierda a derecha.
2. Si el carácter es un operando, se muestra.
3. De otra forma,
    - Si el operador es mayor que el operador anterior en el stack (o contiene un ‘(' o si el stack está vacío), agrégalo.
    - De otra forma, remueve todo los operadores del stack que sean mayores o iguales al operador (para si encuentras un paréntesis)



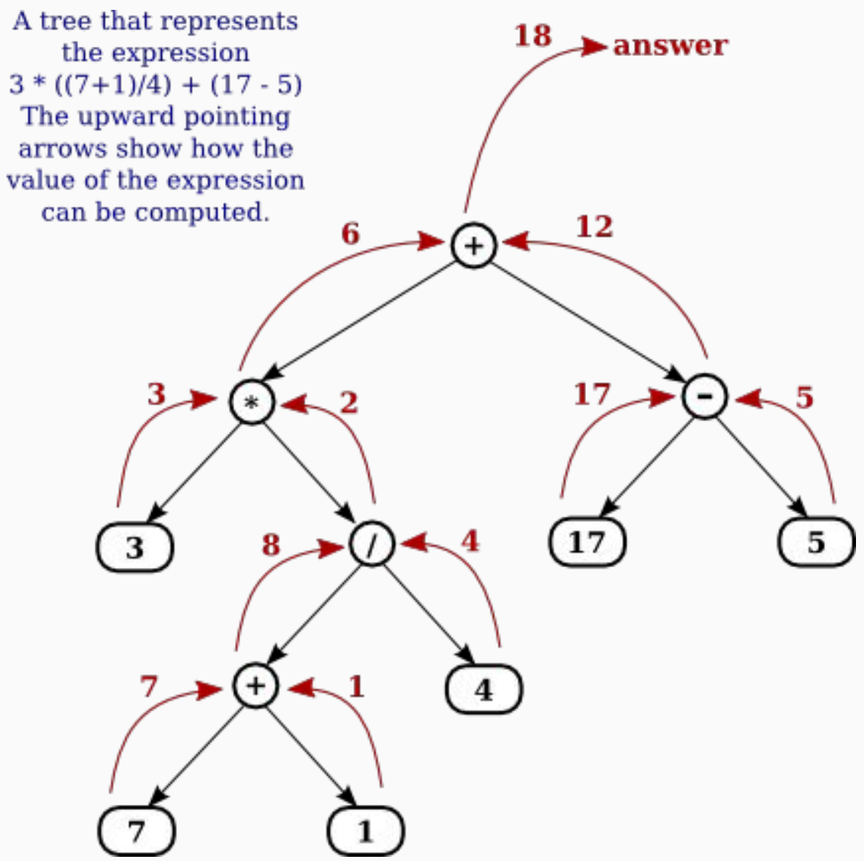
Expression trees

4. Si el carácter es un ‘(‘, agrégalo al stack.

5. Si el carácter es un ‘)’, remueve los elementos del stack hasta encontrar un ‘(‘, incluye ambos paréntesis.

6. Repite los pasos 2-6 hasta el final de la expresión.

7. Remueve los elementos del stack hasta que esté vacío.

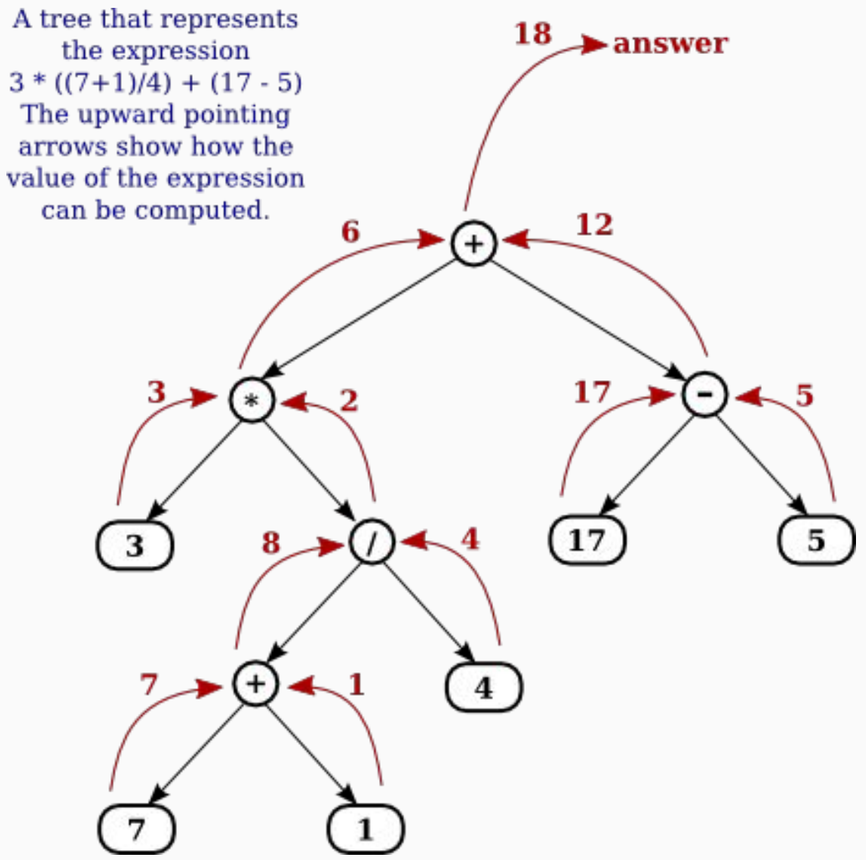


Expression trees

Para procesar el formato postfijo, se empezará a agregar a un stack cada elemento de la expresión de izquierda a derecha.

Cada vez que se encuentre un operador, se operará con los 2 elementos siguientes, y se insertará el resultado en el stack.

Una vez todo la expresión sea procesada, se tendrá la respuesta



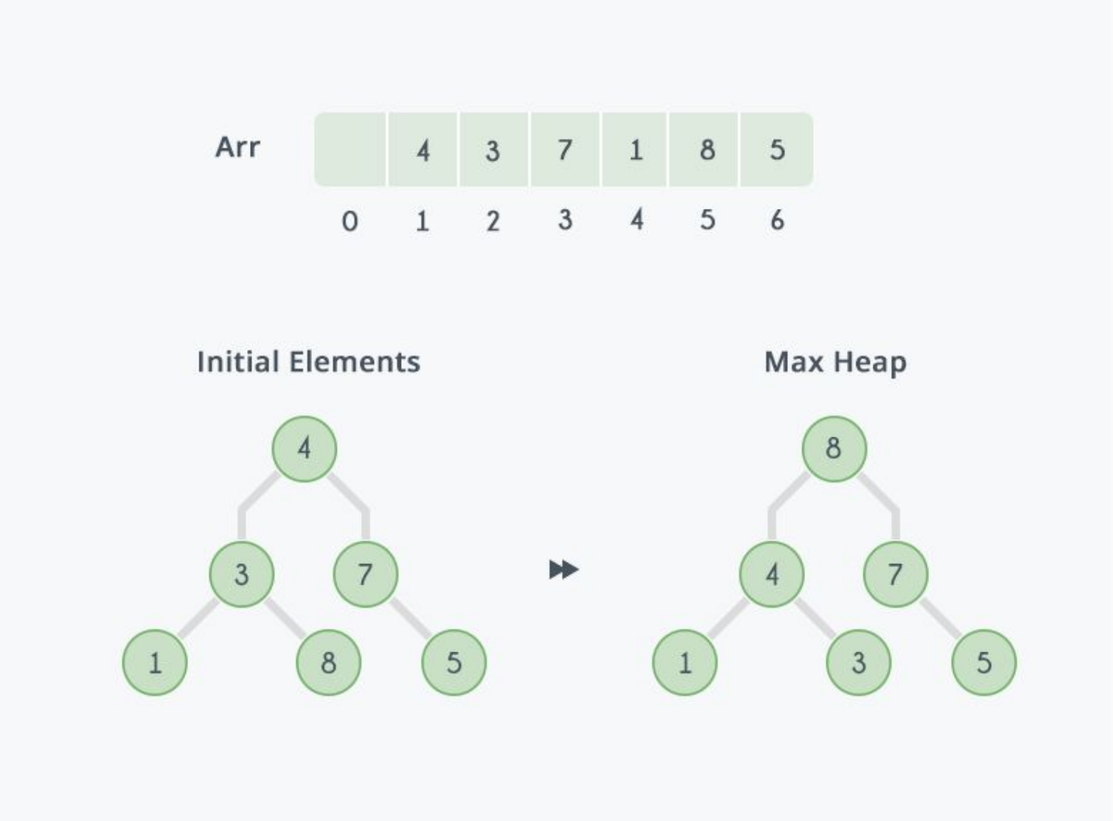
Heaps

Muchas veces necesitamos un acceso rápido al elemento más grande o pequeño de un conjunto de datos.

Por ejemplo:

- Para tener una cola de prioridades
- Ordenar elementos de manera eficiente
- Obtener el K elemento más pequeño o grande de un conjunto

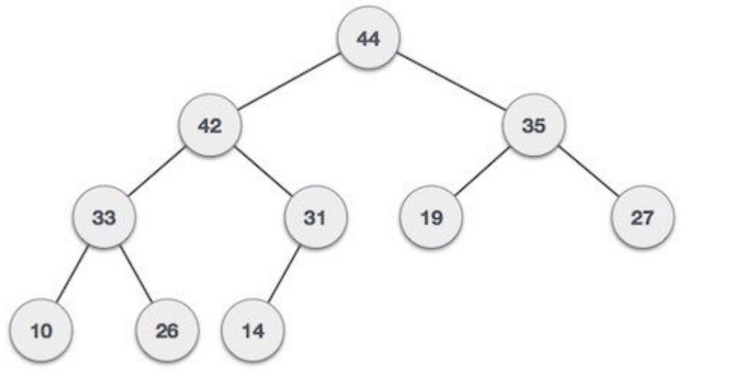
Cómo se les ocurre que podríamos obtener el elemento más grande o pequeño de un arreglo de números?

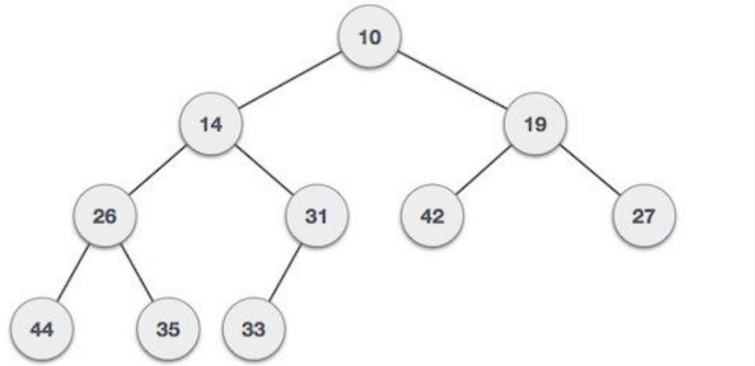


Heaps

Es una estructura de datos tipo árbol binario que satisface ciertas propiedades:

- Es un árbol completo (semi-completo)
- Satisface la propiedad del heap: Para todo nodo, si es **max** entonces el padre es mayor o igual, y si es **min** entonces el padre es menor o igual





En los ejemplos de la derecha cuál sería el max y min heap?

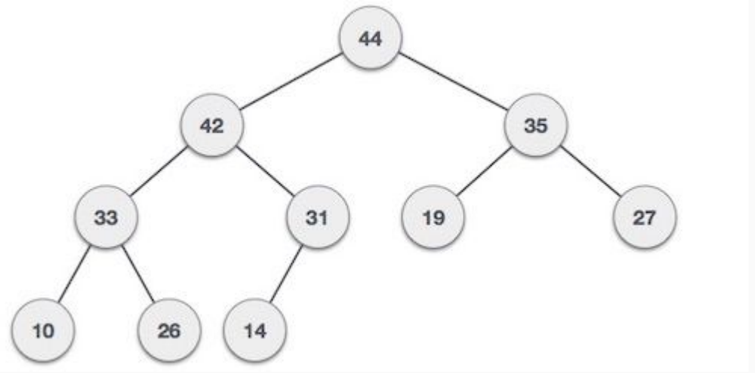
Heaps

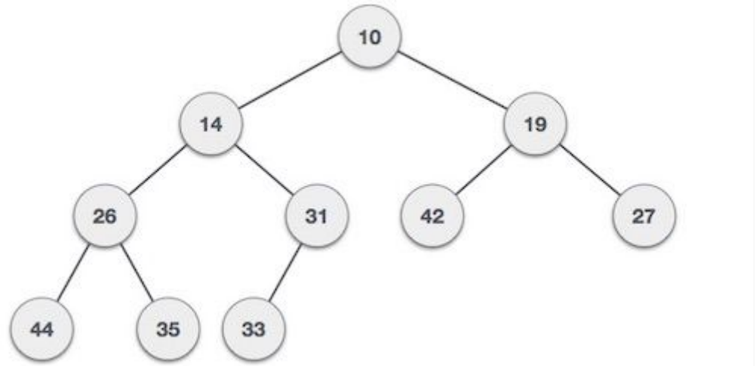
1. **Min heap:** El root siempre es el elemento menor
2. **Max heap:** El root siempre es el elemento mayor

Entonces, cómo creen que se inserta un elemento en un heap?

- Se inserta siempre en el siguiente espacio vacío
- Luego se realiza un **heapify-up** en O(log n), donde el elemento se debe comparar con su padre hasta encontrar su posición correcta

Por ejemplo: Inserten el 38 en el max heap, y 5 en el min heap

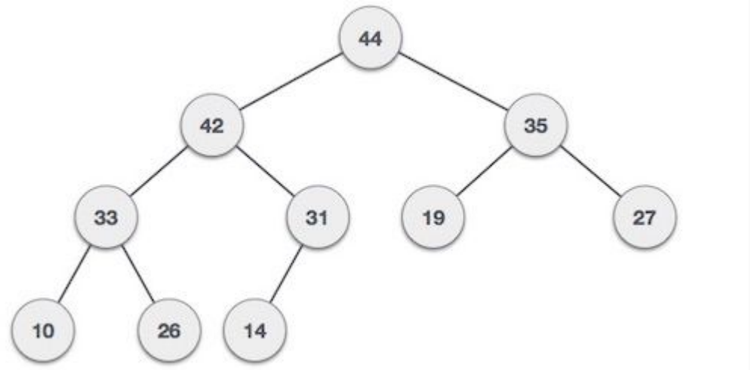




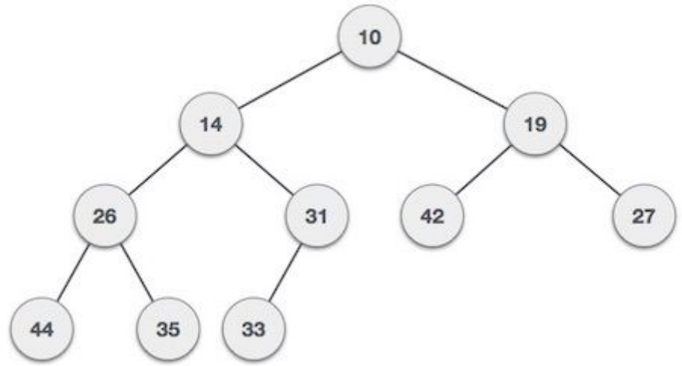
Heaps

Ahora, cómo haríamos el remover?

- Swap del elemento a remover con el último y se remueve el último.
- Se ubica al elemento en su posición correcta usando la función **heapify-down** en O(log n). La función compara y cambia (de ser necesario) al nodo con alguno de sus hijos.



En el caso de un max, se cambiará con el hijo mayor. Min cambiará con el hijo menor. Al hacer un cambio se llamará recursivamente a la función

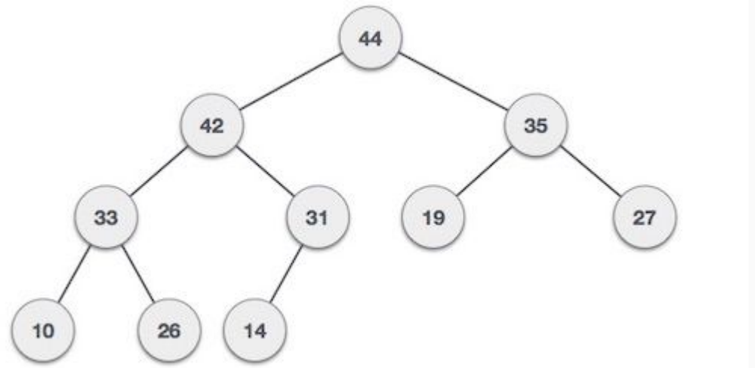


Cuál creen que será el elemento que se eliminará con más frecuencia?

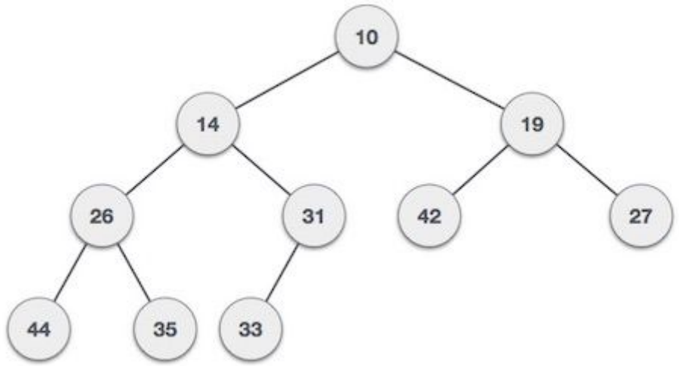
Heaps

Cómo implementarían un Heap?
Quizás con una estructura node?

Lo mejor sería utilizar un simple array. Recuerden la manera en la cual están ordenados los nodos, por ejemplo:



44, 42, 35, 33, 31, 19, 27, 10, 26, 14
10, 14, 19, 26, 31, 42, 27, 44, 35, 33



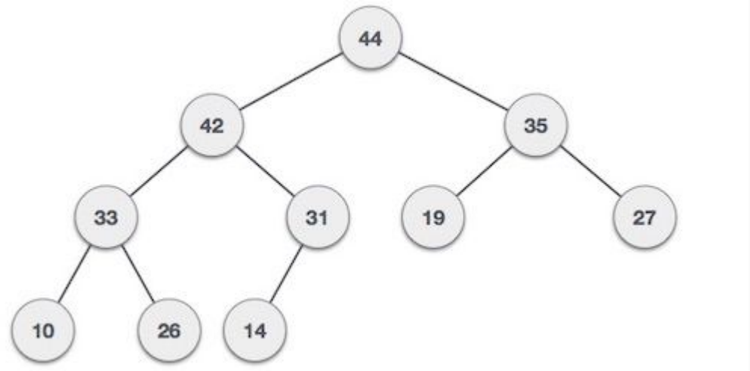
Qué fórmula podrían usar para encontrar los hijos y el padre?

- **Hijo izquierdo:** 2n + 1
- **Hijo derecho:** 2n + 2
- **Padre:** floor((n - 1) / 2)

Heaps

Cómo podríamos saber si un nodo tiene hijo izquierdo o hijo derecho?

Bastaría con validar que el índice del hijo sea menor al tamaño del array

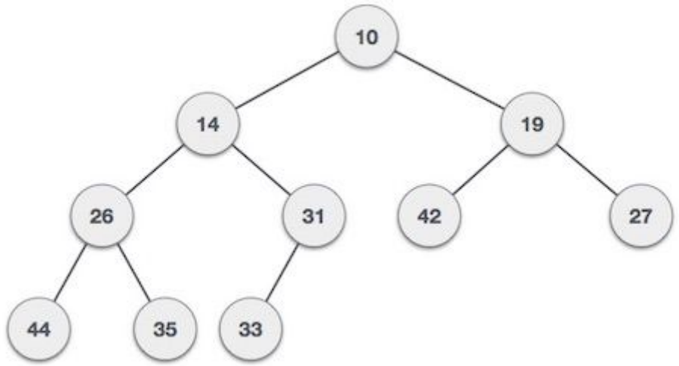


Ejercicios:

Creen un max y un min heap para los siguientes arreglos:

23, 10, 49, 50, 13, 12, 9, 45, 33, 17, 2, 21, 6

70, 12, 15, 17, 66, 54, 48, 59, 24, 13, 33, 61, 4



Heaps

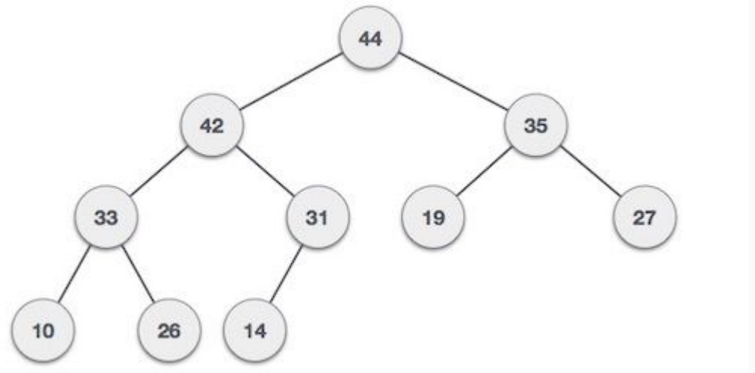
Cómo se construiría un heap desde un arreglo?

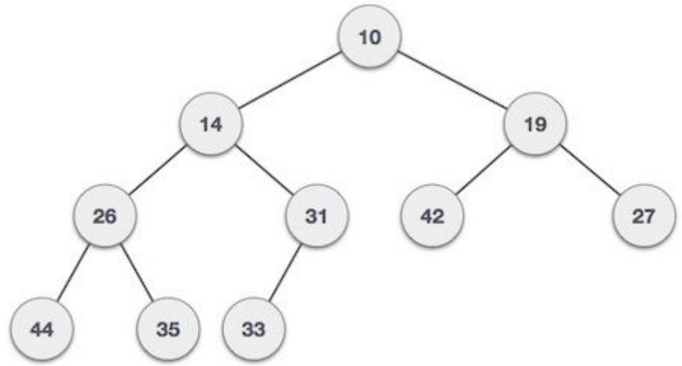
Primero, no sería necesario trabajarlo desde el insertar

- Habría que agregar todos los elementos al heap
- Luego empezar a llamar **max-heapify** O(log n) desde la mitad del árbol hasta el padre. Dependiendo del tipo de heap, max-heapify cambia al mayor/menor de los hijos y lo cambia con el padre.

Por qué desde la mitad del árbol?

Max-heapify se va a llamar n/2 veces. El algoritmo final toma O(n), y max-heapify no siempre toma O(lg n)





Heaps

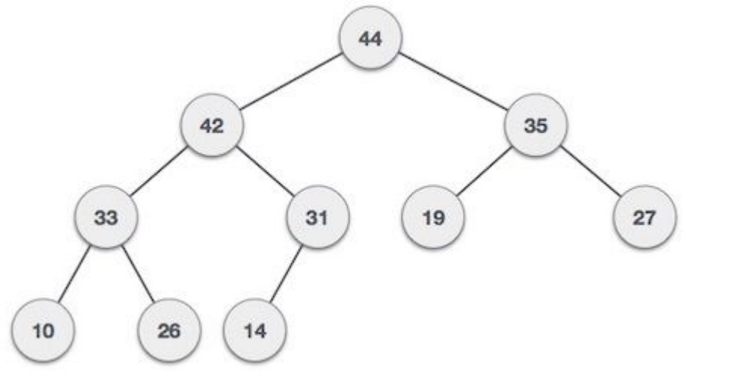
Cuáles serían los usos de los heaps?

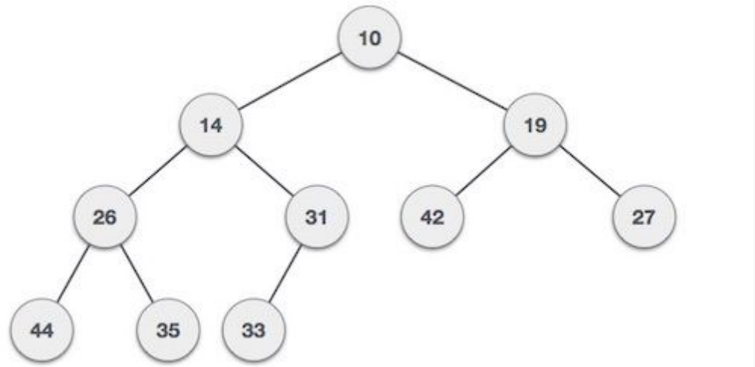
Colas de prioridad, scheduling, prim o kruskal, ordenamientos, etc.

Cuáles son los tiempos de ejecución para sus funciones?

insertar: O(lg n)
remover: O(lg n)
construir el heap: O(n)
encontrar el máximo o mínimo: O(1)

Nota: Recuerden en un árbol casi completo, la altura es aproximadamente lg n.





Heapsort

Cómo usarían los heaps para ordenamiento?

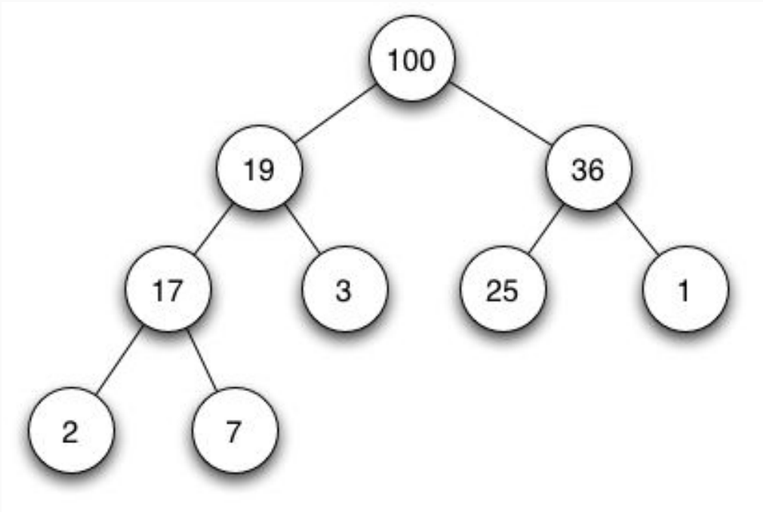
Para el heapsort **primero** creamos un max heap de nuestro array O(n), por ejemplo de: 36, 3, 25, 2, 19, 100, 1, 17, 7

Segundo, removemos nuestro primer elemento del max heap y lo cambiamos por la última posición, quedando el siguiente array O(1):

7, 19, 36, 17, 3, 25, 1, 2, 100

Tercero, volvemos a armar el heap sin el último elemento O(lg n), y se continúa hasta que solo quede un elemento en el heap O(n):

36, 19, 25, 17, 3, 7, 1, 2, 100



100, 19, 36, 17, 3, 25, 1, 2, 7

Heapsort

Ejercicios:

Ordene los siguientes elementos utilizando heapsort, explique el proceso

A. 13, 75, 79, 54, 2, 3 (de mayor a menor)
B. 1, 2, 3, 4, 5, 6 (de menor a mayor)

Conjuntos disjuntos (disjoint sets)

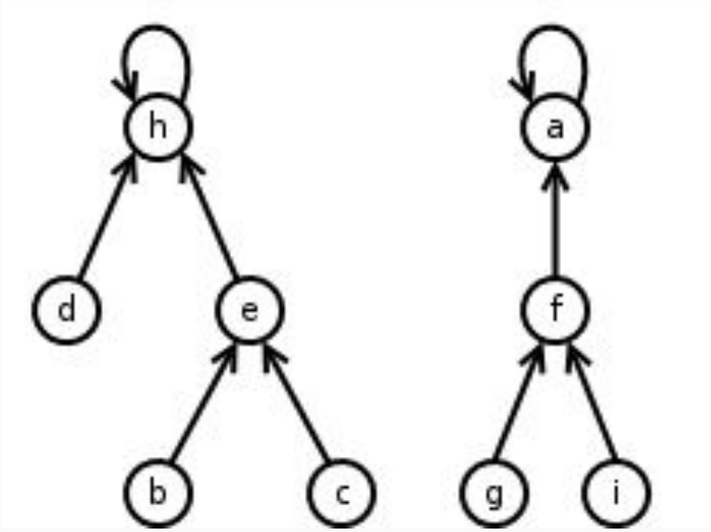
Es una estructura de datos que realiza un seguimiento de todos los elementos que están separados por conjuntos no conectados

Por ejemplo, en el lado derecho tenemos dos disjoint sets:

{h, d, e, b, c} y {a, f, g, i}

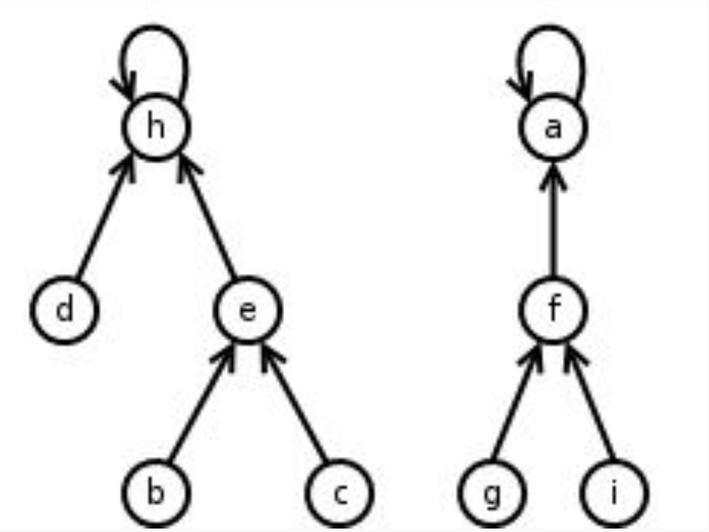
Los Disjoint sets nos pueden ayudar a encontrar componentes conectados, ciclos en un grafo no direccionado, o en la implementación de Kruskal

Los disjoint sets proveen operaciones que se ejecutan en un tiempo casi constante



Conjuntos disjuntos (disjoint sets)

Se elige un representante de cada disjoint set, por ejemplo de los casos de la derecha sería h y a.



Sus principales operaciones son:

1. **MakeSet(x):** Crea un nuevo conjunto
2. **Link(x, y):** Une dos conjuntos con representantes x y y, suponiendo que x != y.
3. **Find(x):** Obtiene el elemento representativo del conjunto que contiene a x

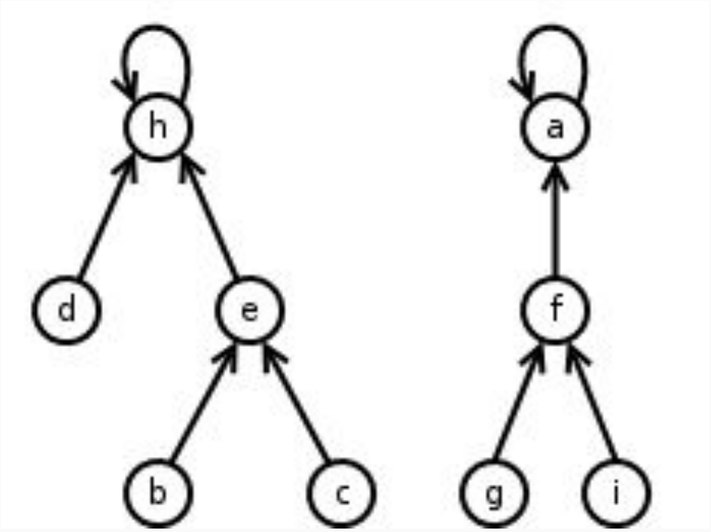
   Podemos unir los conjuntos de la derecha y formar. {h, d, e, b, c, a, f, g, i}

Conjuntos disjuntos (disjoint sets)

```
struct Node {
    Node* parent;
    int rank;
    int data;
}
```

MakeSet se ejecuta en tiempo O(1) para crear un nuevo conjunto con rank 0

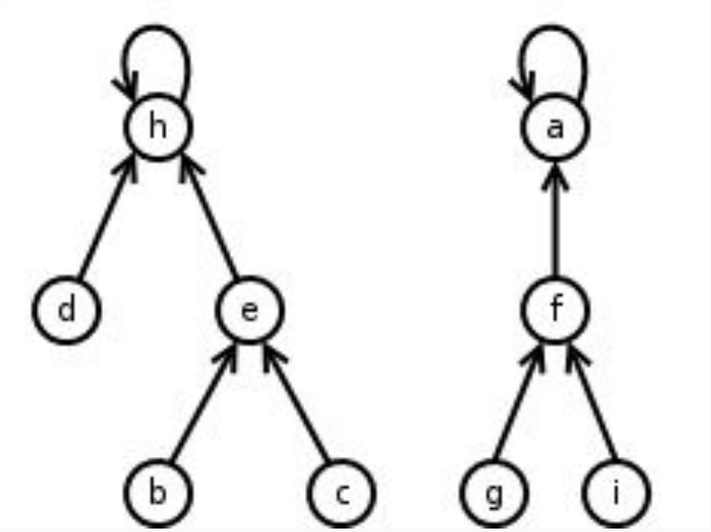
Find va a seguir el camino del padre hasta encontrar un bucle y llegar a la raíz



Conjuntos disjuntos (disjoint sets)

```
struct Node {
    Node* parent;
    int rank;
    int data;
}
```

Link verifica las raíces de dos disjoint sets, si son diferentes une los dos sets. El campo rank se increment en uno, si es igual en ambos disjoint sets, de lo contrario se usa el mayor



Conjuntos disjuntos (disjoint sets)

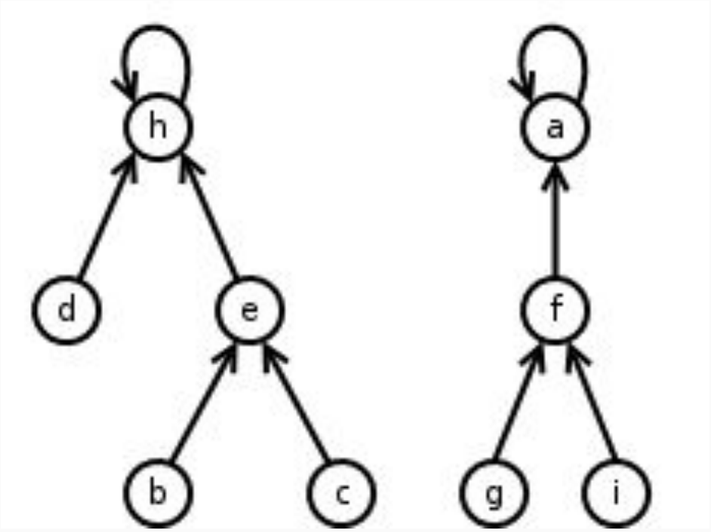
Ejercicios:

Cree un disjoint set con el siguiente arreglo y operaciones

1, 2, 3, 4, 5, 6, 7

makeSet(1)
makeSet(2)
makeSet(3)
makeSet(4)
makeSet(5)
makeSet(6)
makeSet(7)

union(5, 2)
union(2, 3)
union(4, 3)
union(1, 6)
union(7, 1)
union(1, 5)



Conjuntos disjuntos (disjoint sets)

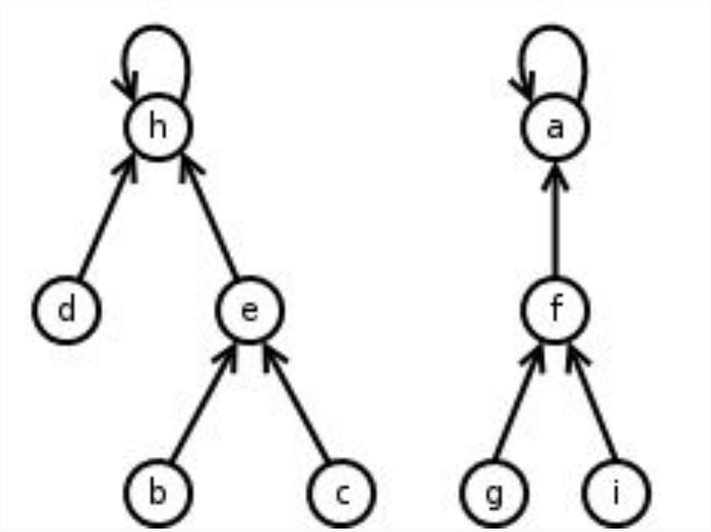
Cuánto tiempo tarda la función find para encontrar al representante del conjunto en el peor caso?

O(n)

Qué se podría hacer para mejorar el tiempo de las funciones?

Path compression! Durante la ejecución de find(x), después de localizar la raíz del árbol que contiene x, se hace que la ruta apunte directamente al representante

Path compression incrementa el tiempo del find en un factor para la primera vez que se realiza sobre un elemento, pero vuelve constantes los accesos posteriores



UTEC
UNIVERSIDAD DE INGENIERÍA
Y TECNOLOGÍA

Welcome to Algorithms and Data Structures! - CS2100