Ayudantia T2

...

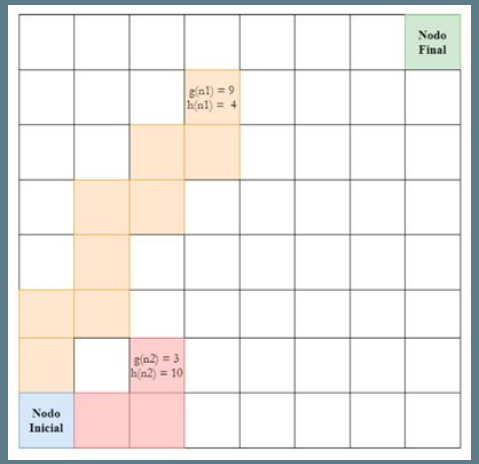
Esteban Brzovic ebrzovic@uc.cl
Susana Figueroa sfigueroa3@uc.cl

Parte 1: Implementación de Weighted A* (2 puntos)

La implementación que mostramos en clases de Weighted A*, y que se encuentra disponible junto a los archivos de la tarea, tiene un defecto. El defecto es que una buena implementación de A* no usa exclusivamente la función $f(n) = g(n) + h(n)$ como prioridad en Open. ¿Por qué? En parte quiero que tú lo descubras en esta pregunta, pero la clave tiene que ver con cómo se rompen empates dentro de Open. Una buena implementación, al tener dos nodos en la Open con el mismo valor f (es decir, dos nodos n_1 y n_2 tales que $f(n_1) = f(n_2)$), debiera preferir para expansión a aquel que tiene el mayor valor g, o, equivalentemente, menor valor h.

1. Explica por qué, intuitivamente, debiéramos usar la regla de quiebre de empates que se describe arriba. En otras palabras, explica por qué esta regla 'tiene sentido'.

2. Explica por qué, al usar esta regla de desempate, Weighted A* sigue siendo w-subóptimo. Para esto, con la mayor claridad posible, explica por qué la demostración del video, sigue funcionando.



Recordar:

- f(n) = g(n)+h(n)
  - g(n) = costo que tomo para llegar a el nodo n.
  - h(n) = el valor de la heurística (costo que le queda para llegar al nodo final).

"Si h es admisible, weighted A* encuentra una solución cuyo costo es a lo más w veces el óptimo (siendo w el peso)."

- Jorge Baier https://www.youtube.com/watch?v=_41v4I5GTNc&ab_channel=JorgeBaier

3. Describe un problema de búsqueda en el cual A* sea *exponencialmente más ineficiente* cuando es usado quebrando empates de la manera contraria, es decir, quebrando empates hacia nodos de menor valor g. Sé lo más precisa posible al mostrar por qué A*, con el rompimiento de empates incorrecto, es tan ineficiente en este caso. **Ayuda:** Acá es conveniente que consideres un problema de búsqueda en el que al desordenar las acciones de una solución obtienes otra solución.

4. Implementa esta regla de quiebre de empates en el archivo base (`Parte1/astar.py`) y reporta (en un gráfico o una tabla) la diferencia en rendimiento (medida en expansiones) de tu 'versión mejorada' al encontrar una solución óptima. Ejecuta el archivo `test_astar.py` solamente sobre los 10 primeros problemas.[1]

5. Finalmente, revisa la ayudantía del lunes 3 de mayo, e implementa el método `estimate_suboptimality`.

Qué entregar en esta parte: En un subdirectorio `Parte1/` un archivo `respuesta.pdf` con la respuesta para 1–4 y el archivo `astar.py` modificado. Antes de hacer commit, asegúrate que leíste la página 1 de este enunciado.

Recordar:

- Comparar la optimalidad con distintos pesos w.
- Solo ejecutar los 10 primeros cuando estás evaluando performance óptimo.
- Astar(init, heuristic, w) (linea 34 test_astar.py)

Parte 2: Heurísticas (2 puntos)

If you want to master something, teach it
– Richard Feynman

1. En la clase sobre propiedades de las heurísticas, vimos la propiedad de *consistencia* de una heurística.

   a) Demuestra que la suma de distancias Manhattan es una heurística consistente para el problema del puzle de 15.

   b) Demuestra que cuando A* usa una heurística consistente, cada estado puede ser expandido a lo más una vez.[2] **Ayuda:** En clases mostramos que hay una relación que se cumple entre el valor f de un padre y un hijo cuando la heurística es consistente. También mostramos que esto implica que algo ocurre con el valor f mínimo en Open durante la ejecución... La demostración (que es muy breve!) sale de pensar cuáles son las implicancias de esto.

Recordar:

Una heurística se dice consistente si

- $h(s) = 0$, para todo $s \in G$
- $h(s) \leq c(s,s') + h(s')$, para todo vecino s' de s. (s=Padres, s′= Hijos)

2. El puzle de 15 con fichas livianas es una variante del puzle de 15 tradicional, en donde mover la pieza n tiene costo $1/n$. La suma de distancias Manhattan no es admisible para este problema. Propón una variante de la distancia Manhattan que sea admisible para este problema y que no sea igual a dividir la distancia Manhattan por 15. Implementa una clase `LightPuzzle` (en un archivo `light_puzzle.py` que posea un método `light_manhattan()` que calcule tu heurística, y un método `manhattan_div_15()` que contenga la heurística Manhattan dividida por 15.

   Evalúa ambas heurísticas en la práctica y reporta tus resultados, en número de expansiones, con el peso tomando valores en 1, 1.5, 2, 3 y 5. Puedes elegir un número de problemas distinto para cada peso, dependiendo de tu capacidad de cómputo.

Recordar:

- h es admisible si, $h(s) \leq h^*(s)$ para todo s
- Cada ficha tiene un costo distinto.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	←	15

costo óptimo = $h^*(s) = 1/15$

Manhattan = $h(s) = 1$

3. La suma de distancias Manhattan es una buena heurística para el puzle de 15, pero no la mejor que se conoce. La heurística de conflictos lineales (*linear-conflicts heuristic*) extiende a la heurística Manhattan agregando lo que define como *conflictos*. La heurística está descrita en un trabajo científico, disponible haciendo click aquí.

   Lamentablemente, no parecen haber videos en youtube que expliquen esta heurística. Haz un video, explicando de qué se trata esta heurística y súbelo a youtube (no hay necesidad de dejarlo público!). Los objetivos del video son:

   a) Que explique brevemente de qué se trata esta heurística.

   b) Que explique la idea de cómo se calcula, y los casos que se muestran en la Figura 3 del paper.

   c) Que explique si esta heurística es o no una relajación, justificando.

   d) Que muestre por qué para el puzle liviano es necesario cambiar la forma en que se computa, mostrando por qué.

   El video debe ser intuitivo y no entrar en más detalles de lo que se pide. Piensa en transmitir bien la idea y que el video sea comprensible para cualquier alumno que haya tomado un curso en donde le hayan enseñado A* con el ejemplo del puzle.

   Los mejores videos recibirán un premio de parte del profesor (no en nota).

https://cursos.canvas.uc.cl/files/3258962/

4. (Opcional, sin puntaje) En el archivo `puzzle.py` ya hay una implementación de la heurística de linear conflicts tomada desde un repositorio github. Prueba qué tanto mejor que Manhattan es en la práctica. (Usa la versión con quiebre de empates de la Parte 1.) ¿Cuál es el efecto de usar pesos?

[2]Dato interesante, aunque irrelevante para esta tarea: La propiedad enunciada acá, no se cumple para todas las heurísticas admisibles, que sí pueden llevar a A* a reexpandir estados. Debido a que esto influye en la eficiencia de la búsqueda, es preferible usar heurísticas consistentes en la práctica.

Recordar:

- El archivo puzzle.py está en la parte 1 de los archivos

5. **Bonus (+0,5 en nota final):** Implementa una versión no trivial y admisible de la heurística linear conflicts para el problema del puzle de 15 con fichas livianas. Explica cómo funciona tu algoritmo en el informe (esto nos servirá para saber que hiciste este bonus), e incluye la implementación en `light_puzzle.py`.

Qué entregar en esta parte: En un subdirectorio `Parte2/` un archivo `respuesta.pdf` con la respuesta para la pregunta 1, un link hacia el video de la pregunta 3, y el archivo `light_puzzle.py`. Antes de hacer commit, asegúrate que leíste la página 1 de este enunciado.

Parte 3: Machine Learning al Rescate (2 puntos)

En esta parte, veremos cómo es posible utilizar una red neuronal (desde ahora, NN) entrenada para decidir 'cómo actuar' en el puzle de 15. Utilizaremos una NN desarrollada por un exalumno de IIC2613, quien en el contexto de una IPRE generó 1.510.673 estados del puzle de 15 al azar, y resolvió en forma óptima cada uno de estos problemas (hasta el mismo estado objetivo), generando así, un conjunto de pares de la forma (s, a), donde s es un estado y a la acción que inicia un camino óptimo hacia el objetivo. Con un subconjunto de esos ejemplos, entrenó una NN que, dado un estado, intenta predecir la acción óptima a ejecutar. La NN que entrenó resultó ser bastante buena, pero no perfecta. ¡Cómo podría serlo, si hay nada menos que $1,04 \cdot 10^{13}$ estados en el espacio de búsqueda del puzle de 15, o sea, vio una porción ínfima del espacio!

En los archivos de base para esta parte, se encuentra implementado un algoritmo de búsqueda, llamado PreferredA*, que utiliza a esta red. Cosas importantes a notar de este algoritmo:

I. A diferencia de A*, utiliza dos colas de prioridades: `preferred` y `open`.

II. Antes de comenzar a iterar la lista `preferred` está vacía mientras que `open` recibe al estado inicial.

III. Ambas colas están ordenadas en base a $g + h$, tal como A*.

IV. Durante el loop principal, las ideas claves son las siguientes:

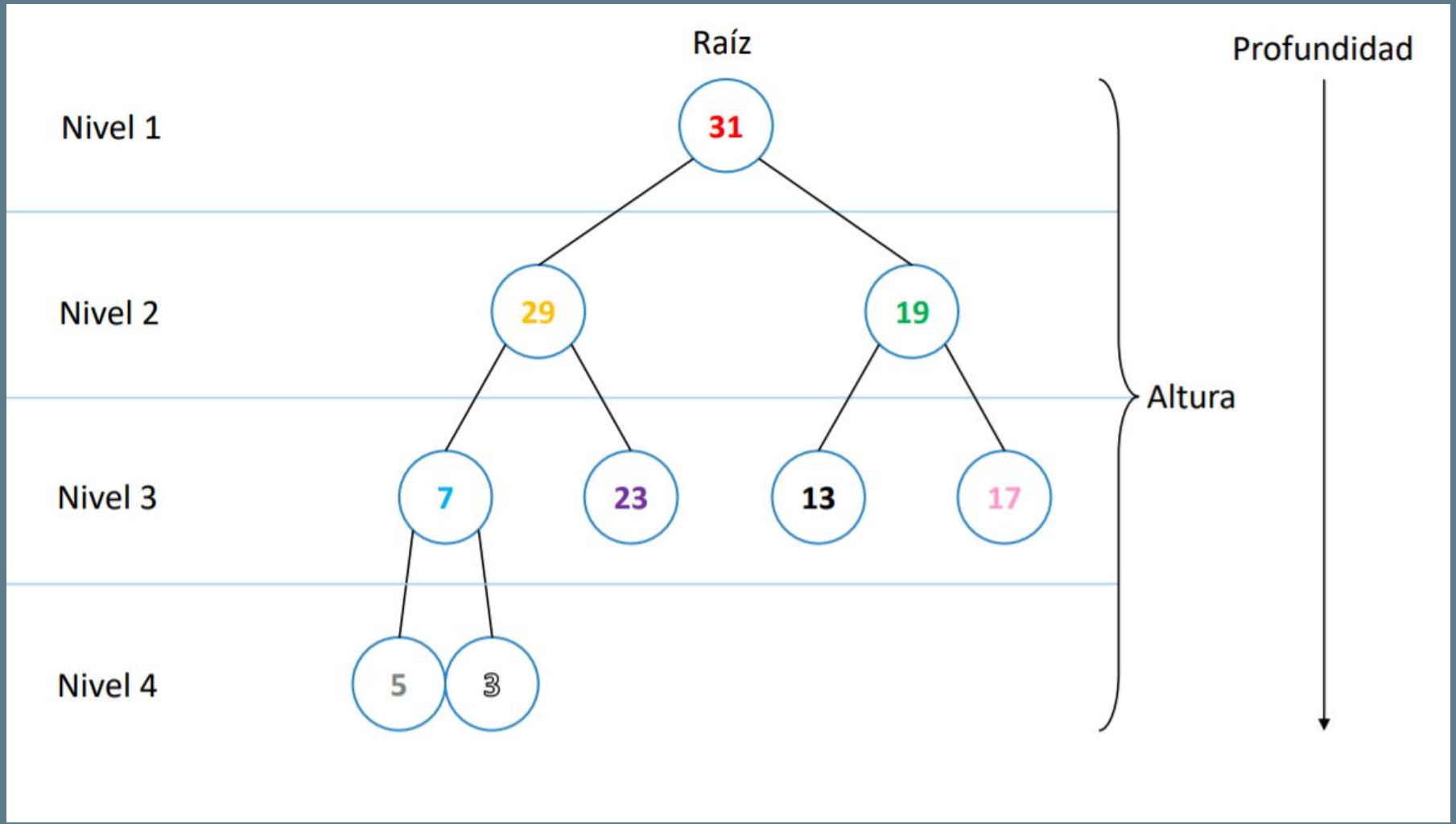
a) El algoritmo alterna entre las dos colas. Esto significa que en algunas iteraciones extrae elementos de una lista y en otras iteraciones, de la otra. Para regular esto usa el contador `counter`, y las variables `self.num_pref` y `self.out_of`. Usando estos parámetros es posible hacer que el algoritmo extraiga más seguido de una u otra cola. Por ejemplo, si `self.num_pref=4` y `self.out_of=10`, entonces podemos esperar que el 40 % de las iteraciones extraiga un nodo de `preferred` y el otro 60 % de `open`. Para entender completamente cómo funcionan estos parámetros, debes ir a mirar el código. ¿Cómo reconoce que un sucesor es el preferido por la NN? Para responder esto, también debes ver el código; específicamente entender lo que está retornando el método `successors`.

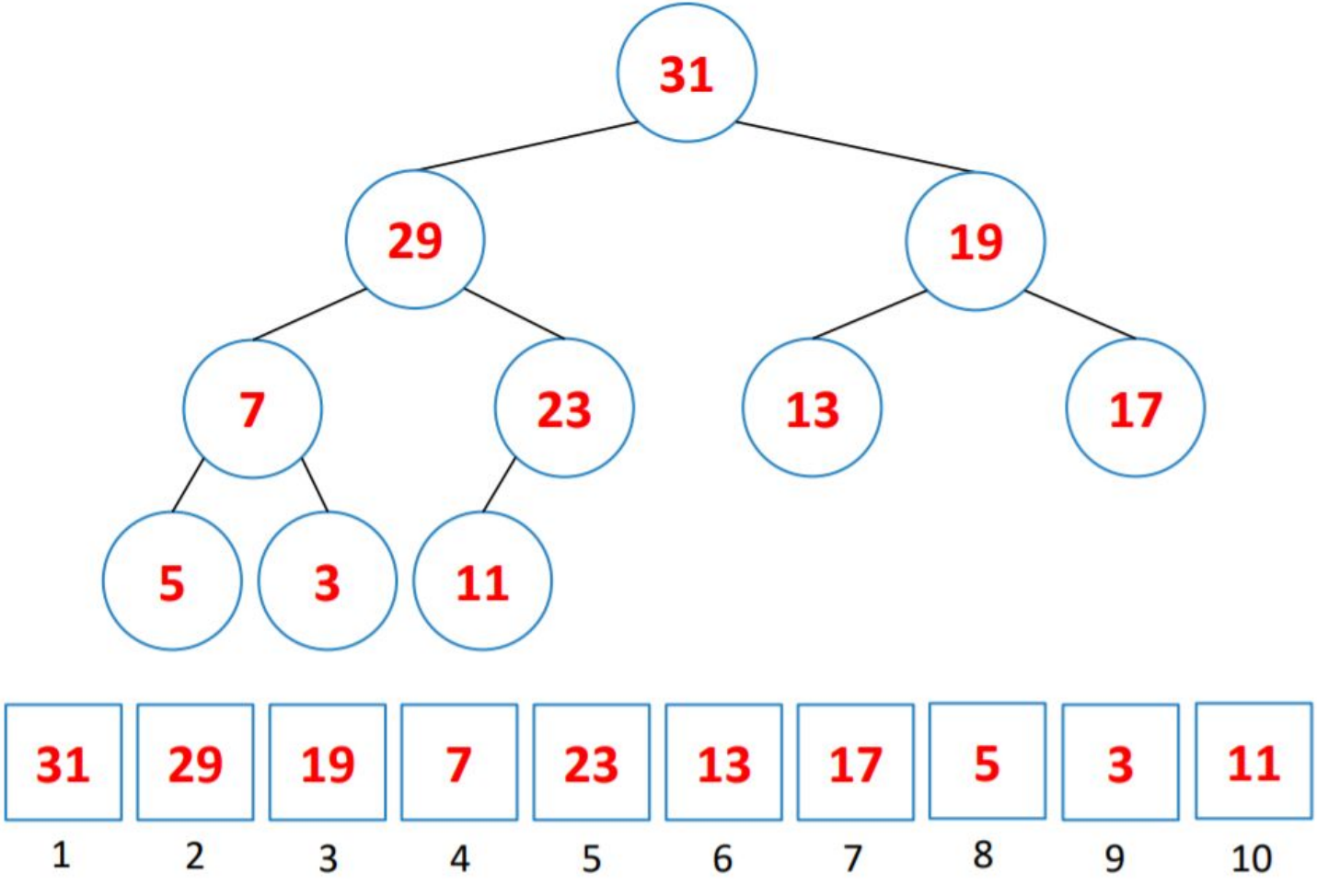
b) Al expandir un estado s, se evalúa el estado a través de la red neuronal, la cual retorna una cierta acción a (que es la que ella 'cree' que lleva a la solución óptima). El sucesor de s que es generado a través de a es agregado a `preferred`, mientras que los demás son agregados a `open`.

- A Partir de un estado s successors escoge los movimientos sucesores y define el preferido según la red NN.
- La elección de las colas están en línea 47-58 de preferred_astar.py
- Succesors se encuentra en la linea 164 de puzzle_nn_new.py

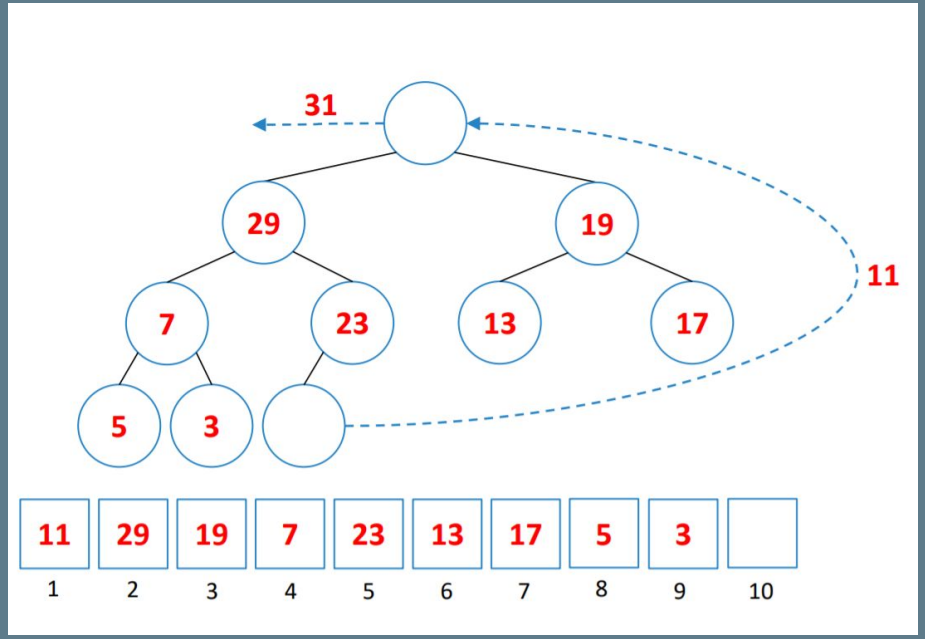
BINARY HEAP

```
def search(self):
    self.start_time = time.process_time()
    self.preferred = MultiBinaryHeap(0)
    self.open = MultiBinaryHeap(1)
```


Raíz
Profundidad
Nivel 1
Nivel 2
Nivel 3
Nivel 4
31
29
19
7
23
13
17
5
3
Altura


31
29
19
7
23
13
17
5
3
11
31 29 19 7 23 13 17 5 3 11
1 2 3 4 5 6 7 8 9 10

Para extraer el nodo 31, esto se hace sacándolo del binaryheap y reemplazandolo por el último nodo.
Este nodo después tiene que hacer un SIFT down hasta que llegue a su posición final .



extract(**H**):

i ← la última celda no vacía de **H**

best ← **H**[1]

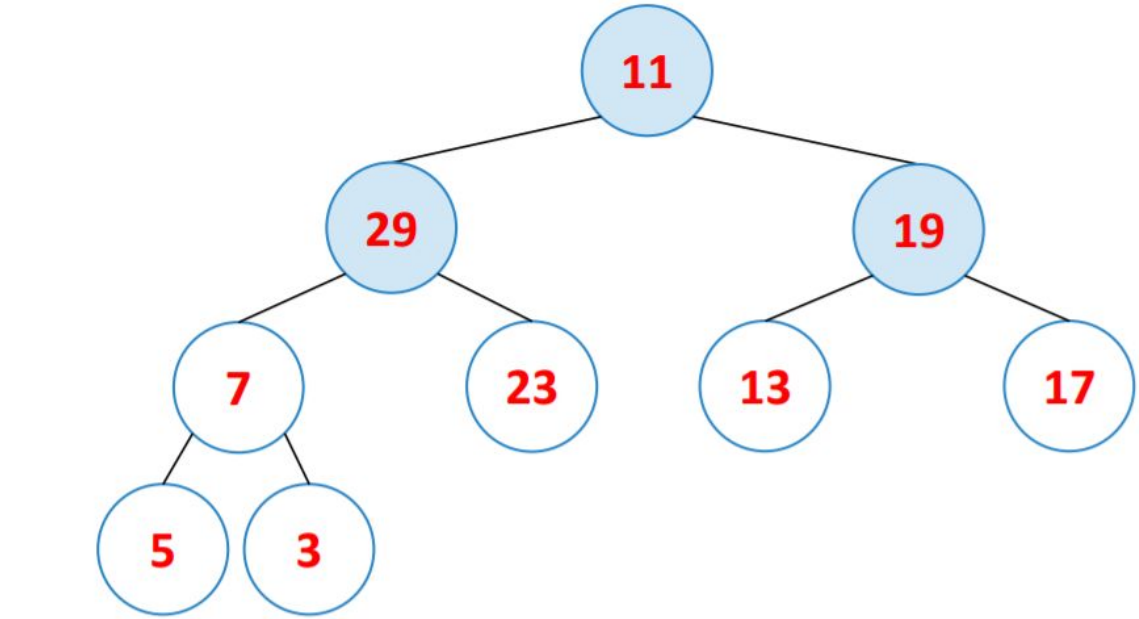
H[1] ← **H**[**i**]

H[**i**] ← ∅

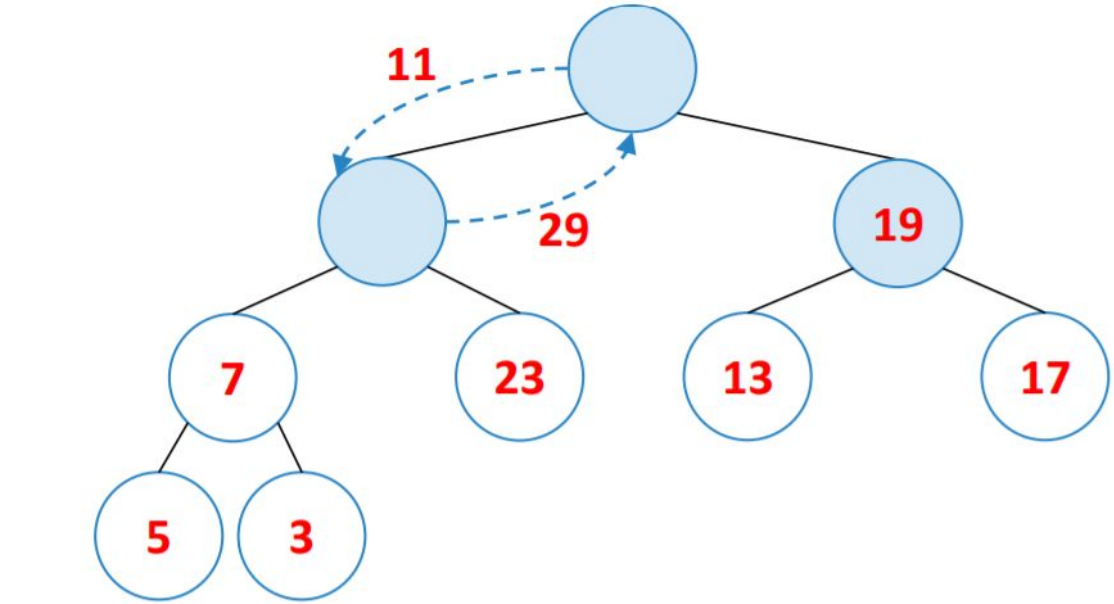
sift down(**H**, 1)

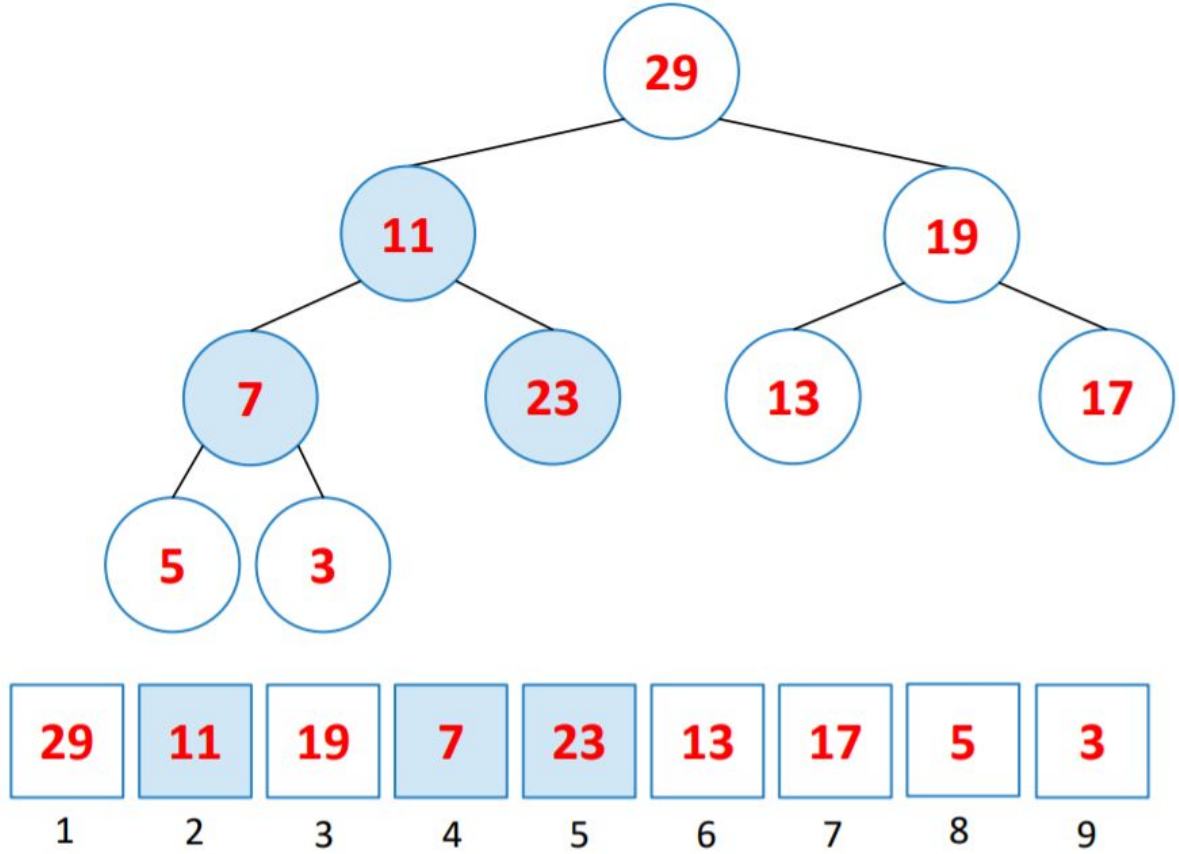
return **best**

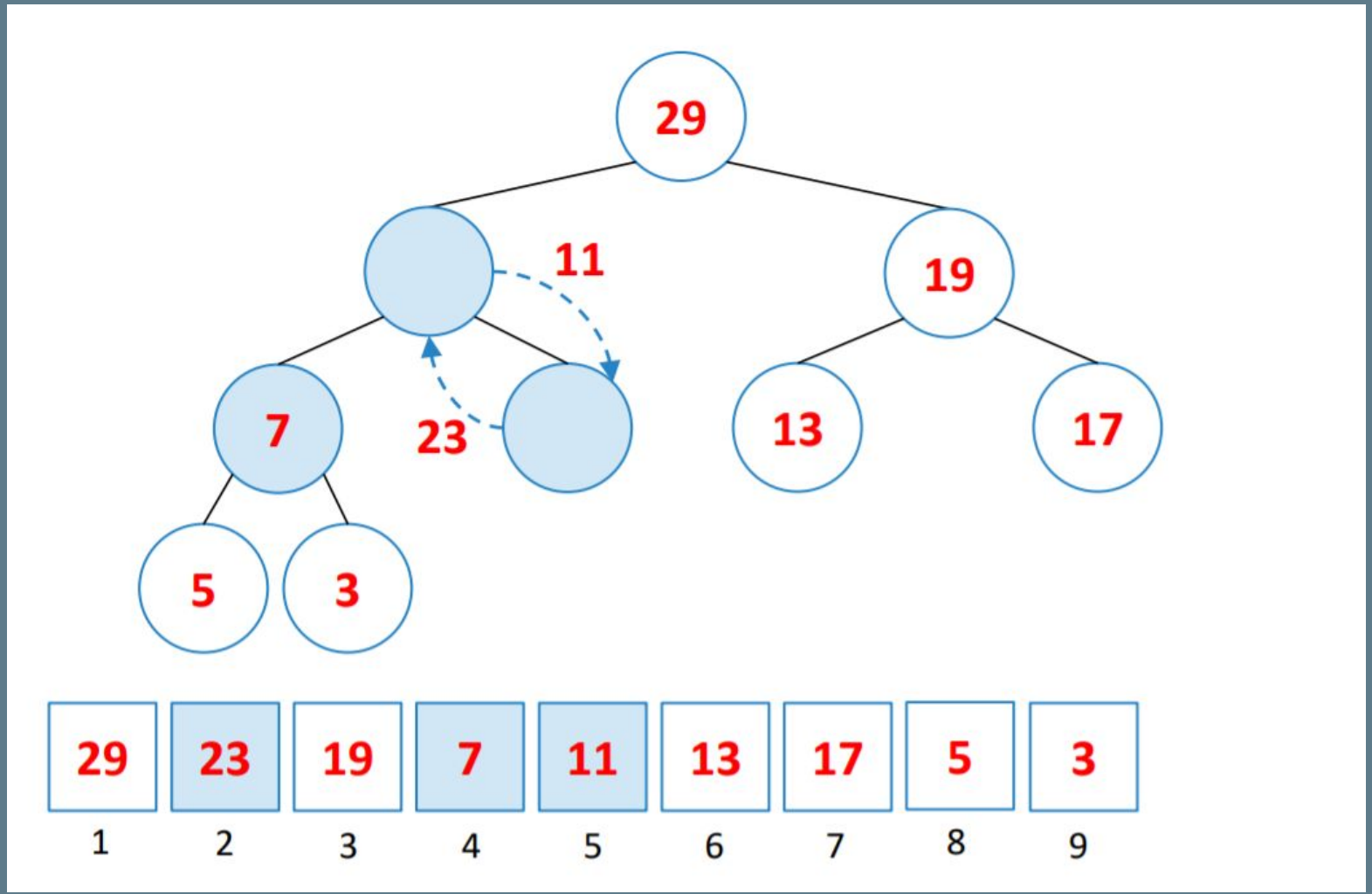
```
sift down(H, i):
    if i tiene hijos:
        i' ← el hijo de i de mayor prioridad
        if H[i'] > H[i]:
            H[i'] ⇄ H[i]
            sift down(H, i')
```

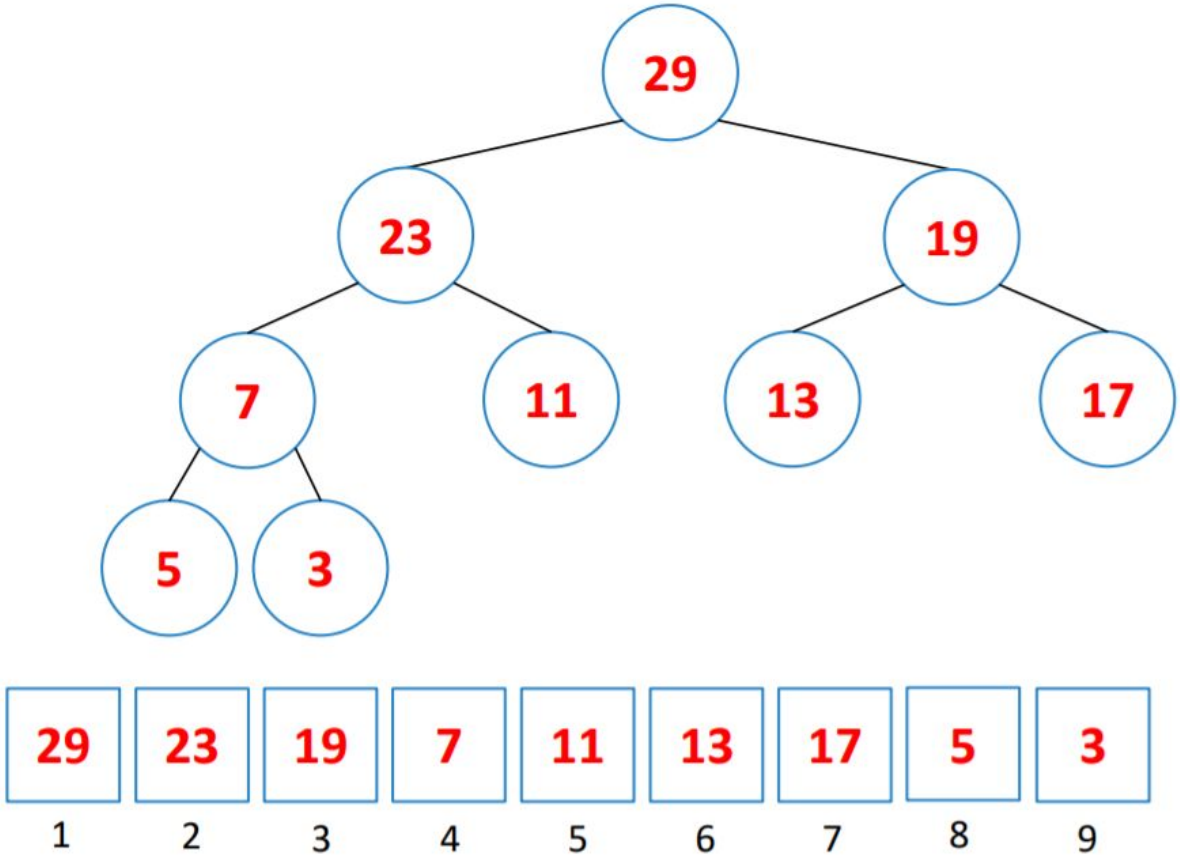


11	29	19	7	23	13	17	5	3
1	2	3	4	5	6	7	8	9


11
29
19
7
23
13
17
5
3
29
11
19
7
23
13
17
5
3
1
2
3
4
5
6
7
8
9


29
11
19
7
23
13
17
5
3
29 11 19 7 23 13 17 5 3
1 2 3 4 5 6 7 8 9


29
11
19
7
23
13
17
5
3
29
23
19
7
11
13
17
5
3
1
2
3
4
5
6
7
8
9


29
23
19
7
11
13
17
5
3
29 23 19 7 11 13 17 5 3
1 2 3 4 5 6 7 8 9

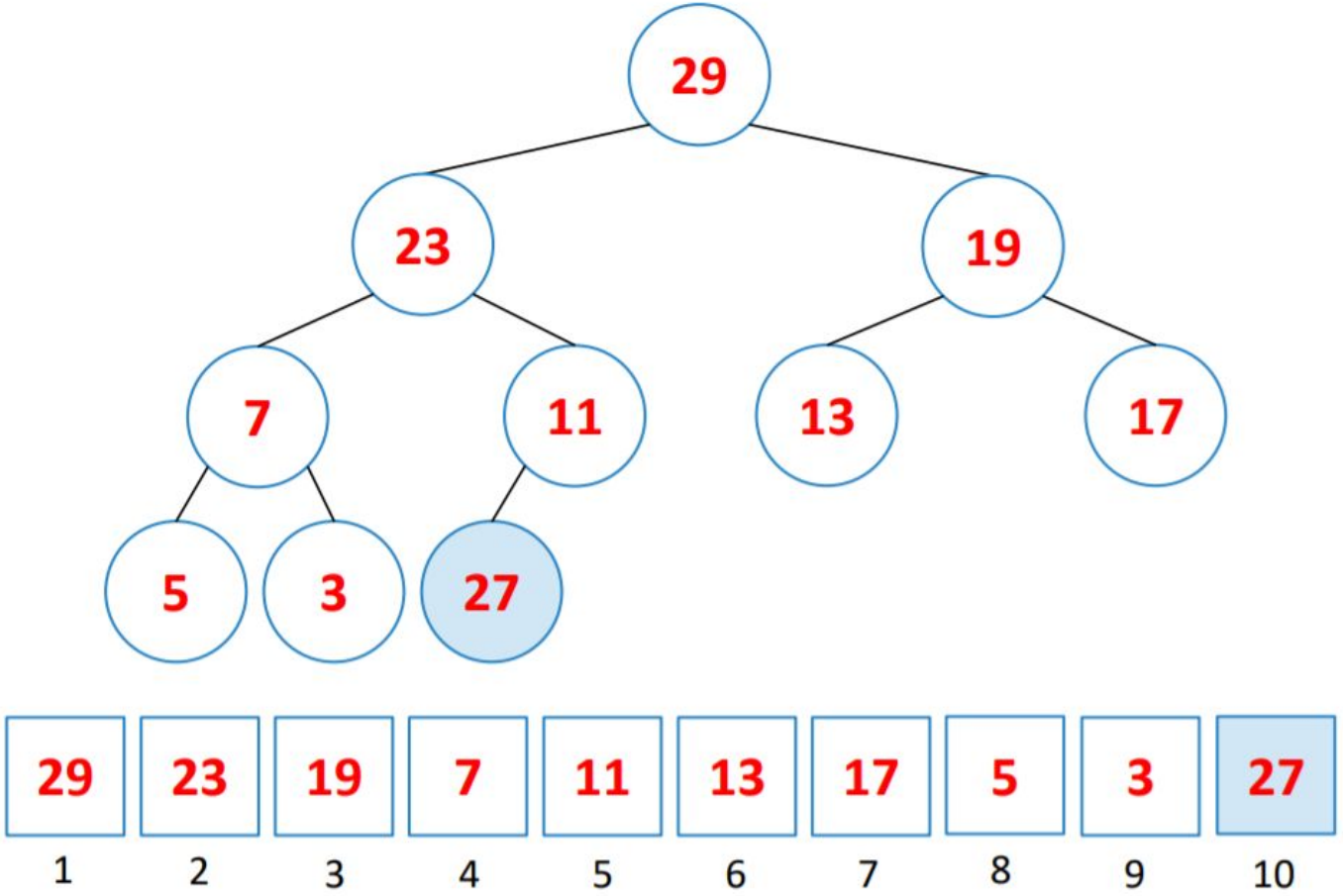
```
insert(H, e):
    i ← la primera celda en blanco de H
    H[i] ← e
    sift up(H, i)
```

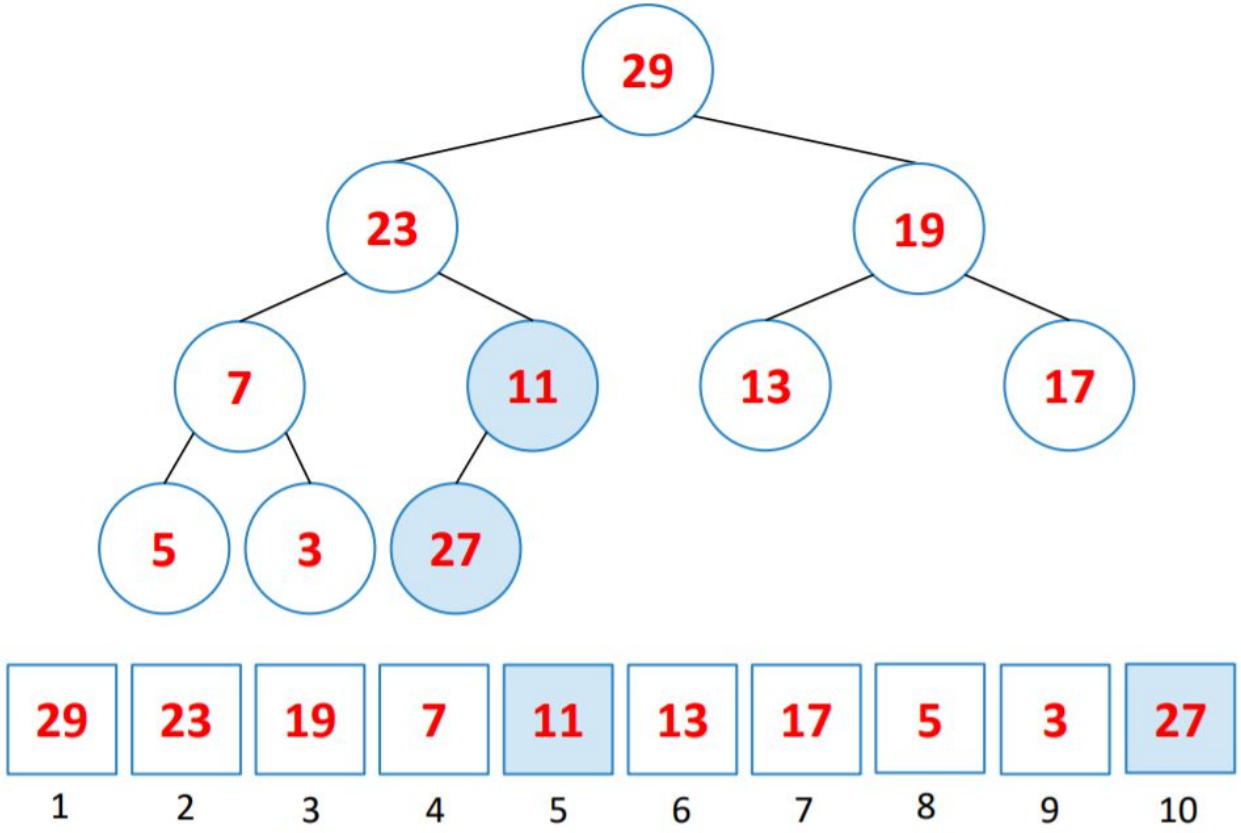
```
sift up(H, i):
    if i tiene padre:
        i' ← el padre de i
        if H[i'] < H[i]:
            H[i'] ⇄ H[i]
            sift up(H, i')
```

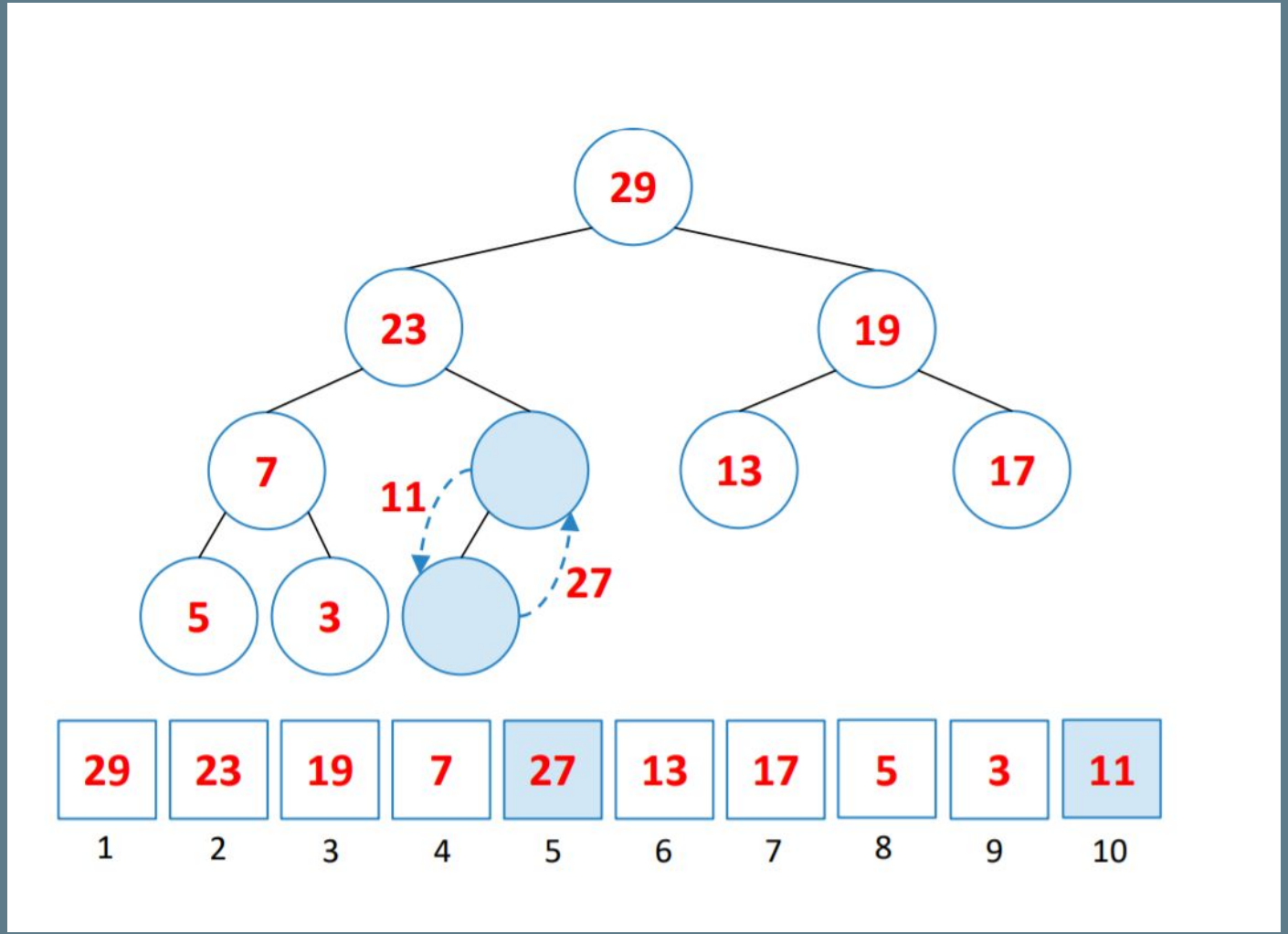
Y como se hace para insertar un nodo?

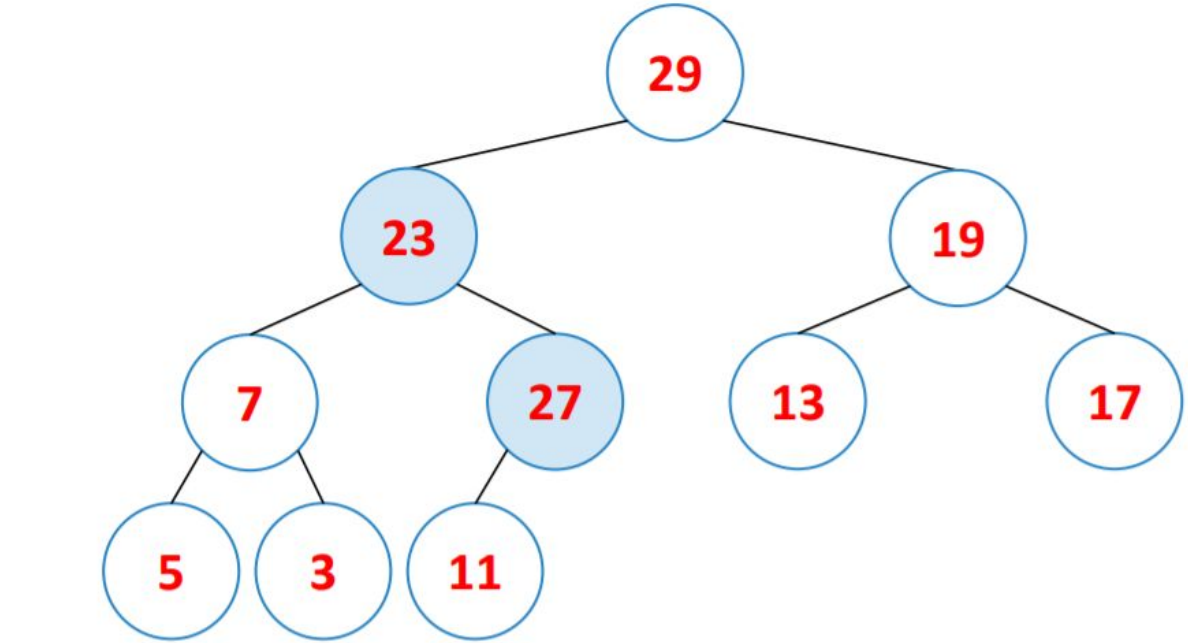
Se coloca en la última posición del binaryheap y desde ahí se hace SIFT up.

Se compara con su nodo padre, si el nodo es mayor a este se intercambian. Se repite hasta llegar a su posición final.

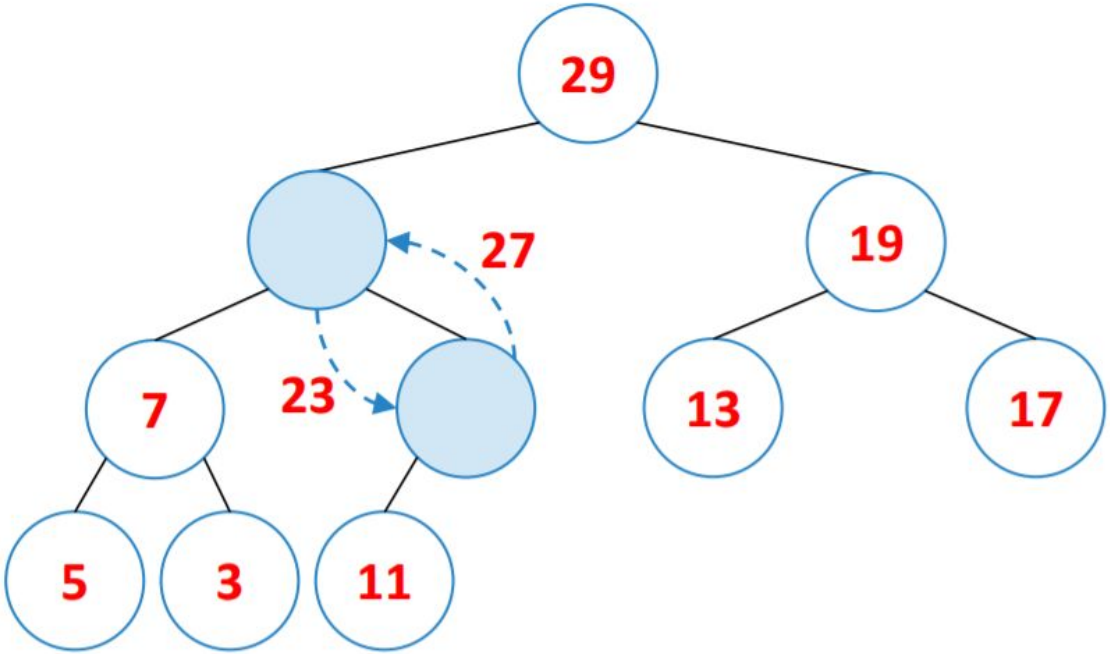

29
23
19
7
11
13
17
5
3
27
29 23 19 7 11 13 17 5 3 27
1 2 3 4 5 6 7 8 9 10


29
23
19
7
11
13
17
5
3
27
29 23 19 7 11 13 17 5 3 27
1 2 3 4 5 6 7 8 9 10

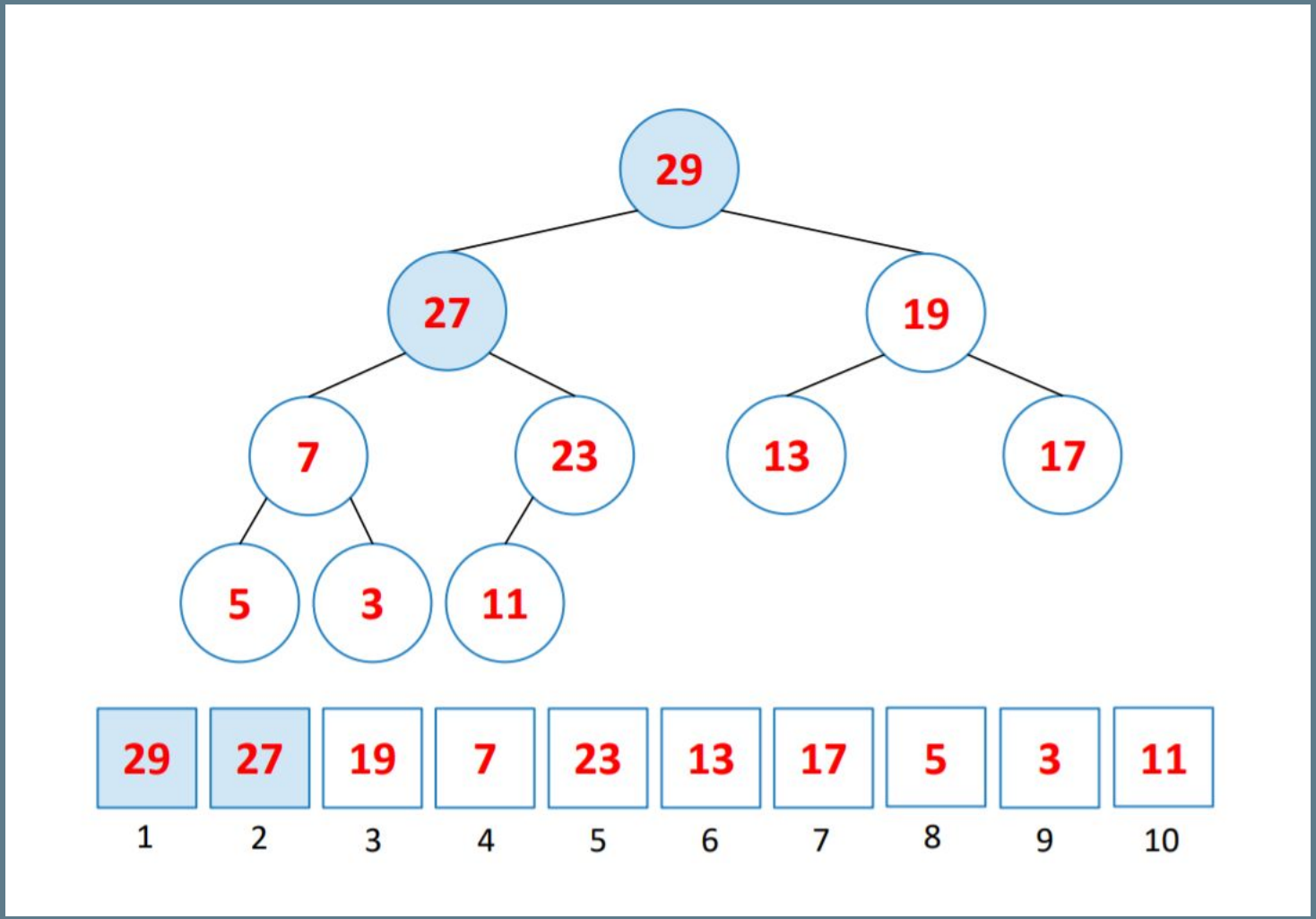

29
23
19
7
11
13
17
5
3
27
29 23 19 7 27 13 17 5 3 11
1 2 3 4 5 6 7 8 9 10


29
23
19
7
27
13
17
5
3
11


29
23
19
7
27
13
17
5
3
11
1
2
3
4
5
6
7
8
9
10



29	27	19	7	23	13	17	5	3	11
1	2	3	4	5	6	7	8	9	10


29
27
19
7
23
13
17
5
3
11
29 27 19 7 23 13 17 5 3 11
1 2 3 4 5 6 7 8 9 10

1. Realiza un estudio experimental, que permita entender cuál es la combinación `num_pref` y `out_of` que tiene el mejor rendimiento experimental, medido en términos de expansiones totales. Intenta y reporta el resultado que obtienes con al menos 6 y a lo más 8 combinaciones de valores, usando la granularidad que prefieras. Sé hábil al reportar; por ejemplo, pon un tiempo límite razonable[3] para las ejecuciones muy largas. Intenta probar con valores extremos e intermedios. Reporta resultados e identifica la mejor configuración. Argumenta si crees que es suficiente con los experimentos que hiciste para llegar a la mejor configuración.
2. Implementa el método `estimate_suboptimality`.
3. Demuestra que el método `estimate_suboptimality` que programaste entrega una cota correcta de suboptimalidad.

Para la 1.

- Esto se encuentra en la linea 8 de preferred_astar.py
- s = PrefAstar(init, heuristic, 9, 10, 1.6) (linea 32 test_preferred_astar.py)

4. Modifica PreferredA* para que dé una garantía de suboptimalidad; es decir, que dado un valor específico de suboptimalidad—por ejemplo 1,5—retorne una solución que no exceda en 1,5 el valor del costo óptimo. La idea clave de esta modificación es que tu algoritmo ya no retornará una solución a menos que pueda demostrar que la solución satisface la cota de suboptimalidad.

   Supongamos que w es el parámetro de suboptimalidad. La modificación es la siguiente. Después de cada extracción de un elemento desde alguna cola, en tiempo constante,[4] calcula el mínimo de $g(s) + h(s)$ para todo s que está en la unión de `preferred` y `open`. Llamemos a ese mínimo m. Sea ahora t el estado recién extraído. Si $g(t)+h(t) > w \cdot m$, entonces insertamos t a `open` y continuamos a la siguiente iteración *sin expandir* a t. La condición de retorno del algoritmo también debes cambiarla, pero te dejo

5. ¿Por qué el algoritmo de arriba garantiza que la solución retornada satisface la cota de suboptimalidad?

6. Compara experimentalmente el rendimiento de tu algoritmo con Weighted A* de la parte 1, para distintas cotas de suboptimalidad (es decir, usando el mismo valor para el peso en Weighted A* y la cota de suboptimalidad).

7. **Bonus (+0,5 en la nota final)**. Supón que una secuencia de estados $\sigma = s_1 s_2 \dots s_n$ es generada a partir de una secuencia de acciones $a_1 \dots a_{n-1}$. Diremos que la discrepancia de σ se calcula sumando 1 por cada vez que a_i *no es* la acción que la NN prefiere, con i moviéndose entre 1 y $n-1$. Implementa el algoritmo `BestFirstDiscrepancy`, que tiene una sola open, pero que ordena la open por discrepancia y quiebra empates por... mejor, piénsalo tú. Observa su rendimiento en la práctica y discute (pero no implementes!) cómo es posible hacer que garantice que la solución retornada satisface una cota dada de suboptimalidad.

Qué entregar en esta parte: En un subdirectorio `Parte3/` un archivo `respuesta.pdf` con la respuesta para las preguntas teóricas, y un archivo `preferred_astar.py` que implementa los cambios pedidos. Antes de hacer commit, asegúrate que leíste la página 1 de este enunciado.