Chapter 19: Fibonacci Heap I

About this lecture

- Introduce Fibonacci Heap
  - another example of mergeable heap
  - no good worst-case guarantee for any operation (except Insert/Make-Heap)
  - excellent amortized cost to perform each operation

Fibonacci Heap

- Like binomial heap, Fibonacci heap consists of a set of min-heap ordered component trees

- However, unlike binomial heap, it has
  - no limit on #trees (up to $O(n)$), and
  - no limit on height of a tree (up to $O(n)$)

Fibonacci Heap

- Consequently,

  Find-Min, Extract-Min, Union,
  Decrease-Key, Delete

  all have worst-case $O(n)$ running time

- However, in the amortized sense, each operation performs very quickly …

Comparison of Three Heaps

	Binary (worst-case)	Binomial (worst-case)	Fibonacci (amortized)
Make-Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Find-Min	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$
Extract-Min	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Insert	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
Delete	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Decrease-Key	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$
Union	$\Theta(n)$	$\Theta(\log n)$	$\Theta(1)$

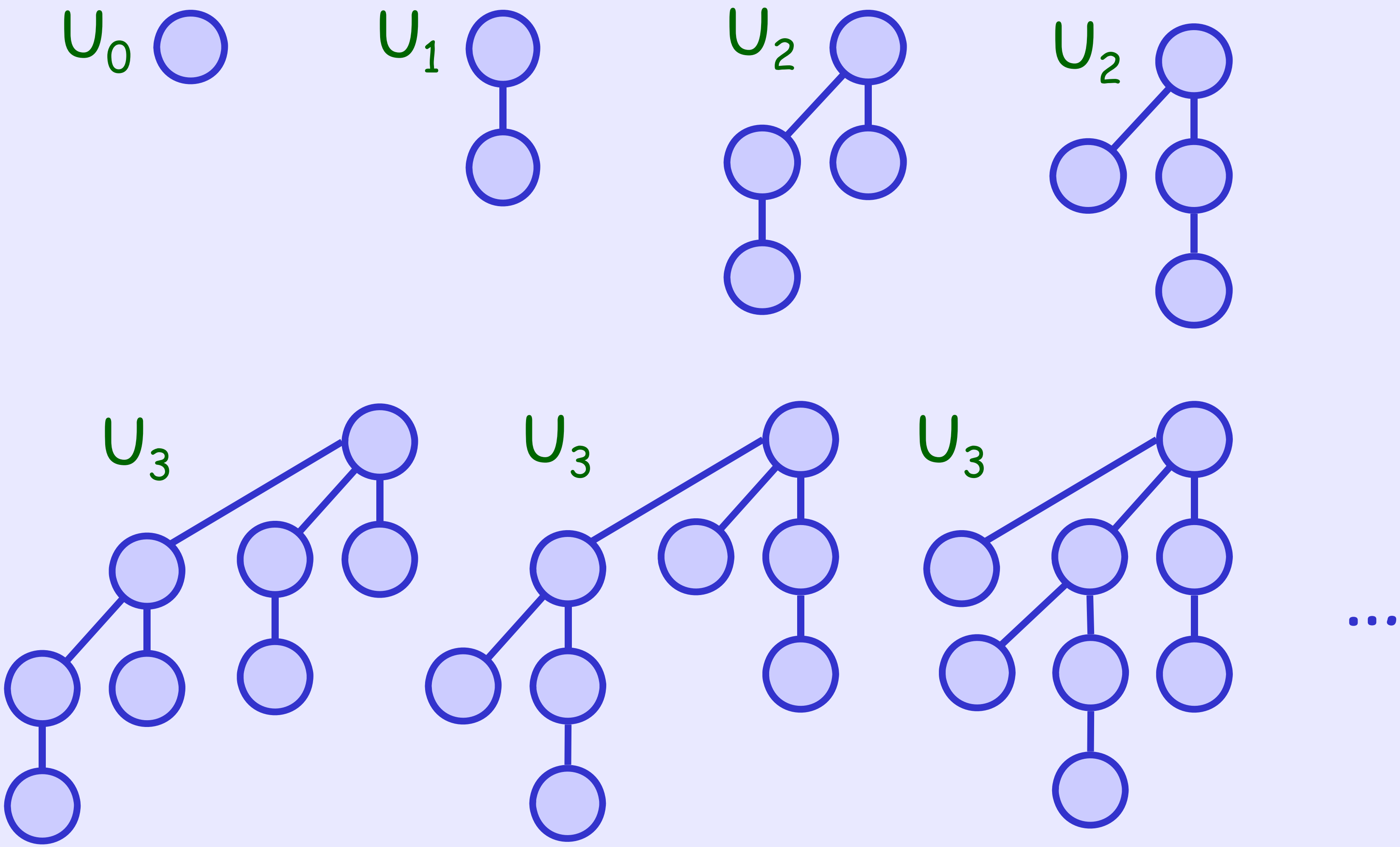
Fibonacci Heap

- If we never perform Decrease-Key or Delete, each component tree of Fibonacci heap will be an unordered binomial tree
  - An order-k unordered binomial tree U_k is a tree whose root is connected to U_{k-1}, U_{k-2}, …, U_0, in any order

➔ in this case, height = $O(\log n)$

- In general, the tree can be very skew

Unordered Binomial Tree



Properties of U_k

Lemma: For an unordered binomial tree U_k,

1. There are 2^k nodes
2. height = k
3. deg(root) = k ; deg(other node) $<$ k
4. Children of root are U_{k-1}, U_{k-2}, …, U_0 in any order
5. Exactly C(k,i) nodes at depth i

How to prove? (By induction on k)

Potential Function

- To help the running time analysis, we may mark a tree node from time to time
  - Roughly, we mark a node if it has lost a child
- For a heap H, let

  t(H) = #trees, m(H) = #marked nodes
- The potential function Φ for H is simply:

$$\Phi(H) = t(H) + 2\, m(H)$$

[Here, we assume a unit of potential is large enough to pay for any constant amount of work]

Remark

- Let Φ_i = potential after i^{th} operation
  ➔ $\Phi_0 = 0$, $\Phi_i \geq \Phi_0$ for all i
  So, if each operation sets its amortized cost α_i by the formula ($\alpha_i = c_i + \Phi_i - \Phi_{i-1}$)
  ➔ total amortized ≥ total actual
- We claim that we can compute MaxDeg(n), which can bound max degree of any node. Also, MaxDeg(n) = $O(\log n)$
  ➔ This claim will be proven later

Fibonacci Heap Operation

- Make-Heap( ):

  It just creates an empty heap

  ➔ no trees and no nodes at all !!

  ➔ amortized cost = O(1)

Fibonacci Heap Operation

- Find-Min(H):

  The heap H always maintain a pointer min(H) which points at the node with minimum key

  ➔ actual cost = 1

  ➔ no change in t(H) and m(H)

  ➔ amortized cost = O(1)

Fibonacci Heap Operation

- Insert(H,x,k):

  It adds a tree with a single node to H, storing the item x with key k

  Also, update min(H) if necessary

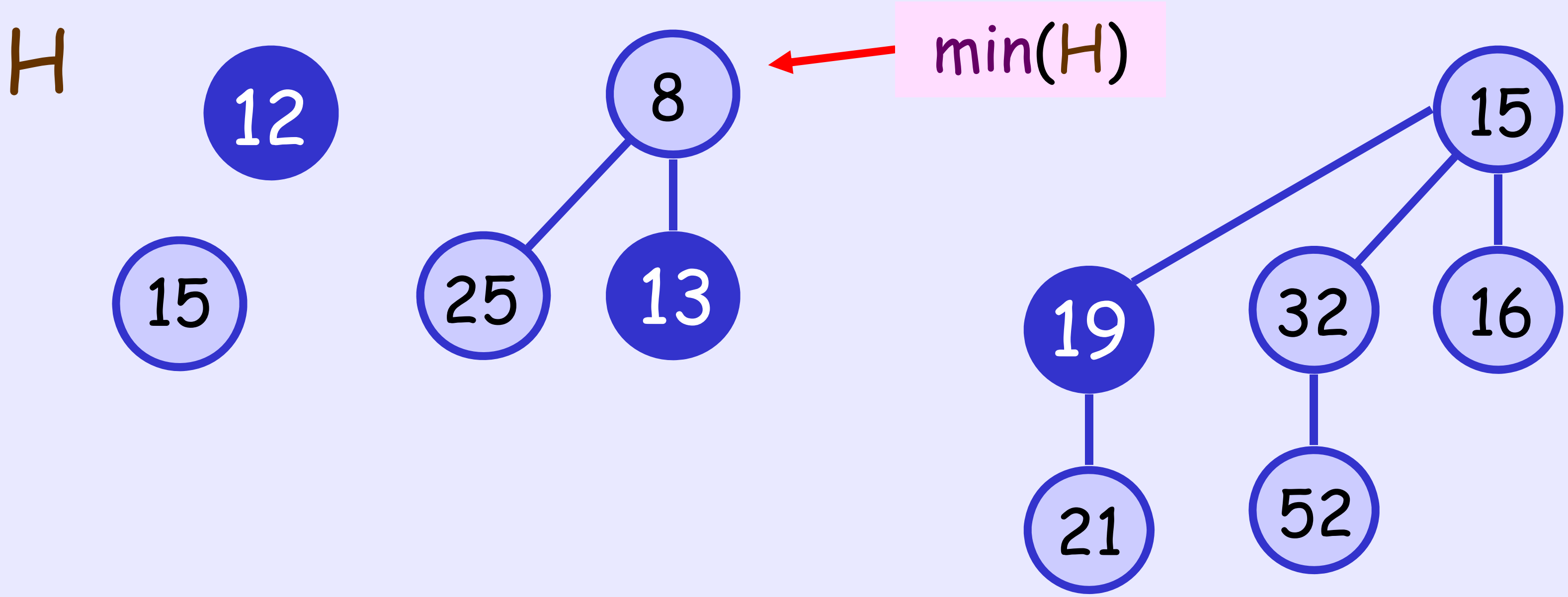
  ➔ t(H) increased by 1, m(H) unchanged

  ➔ amortized cost = 2 + 1 = O(1)

  Add a node, and update min(H)

Insertion (Example)

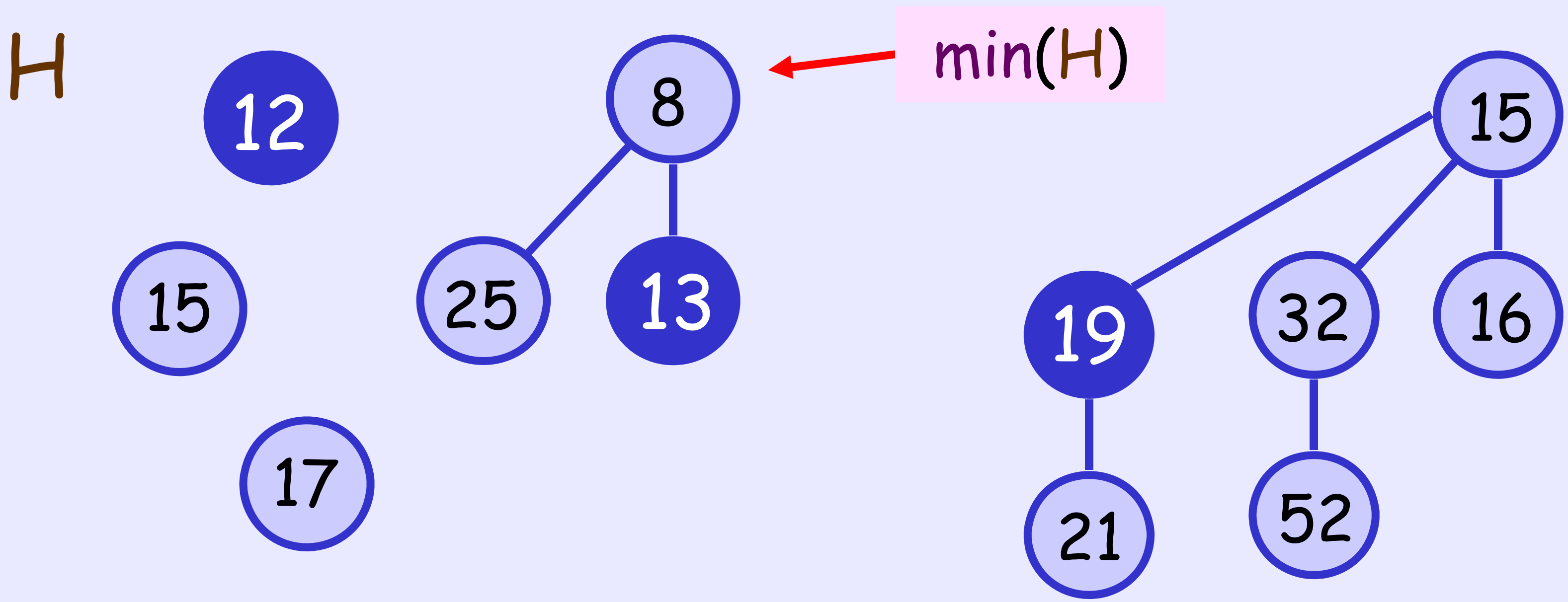
Before Insertion



Marked node

Insertion (Example)

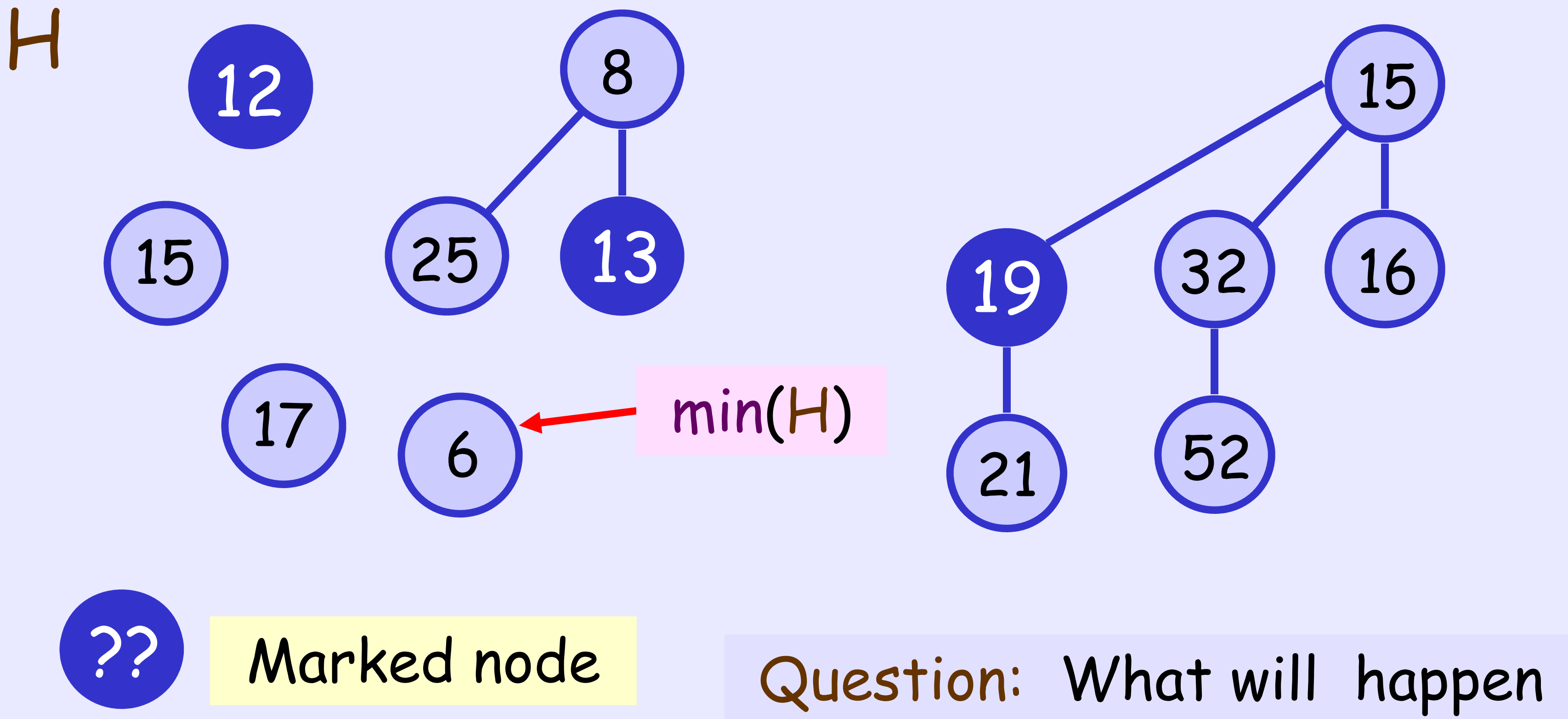
Inserting an item with key = 17



Marked node

Insertion (Example)

Inserting an item with key = 6



Question: What will happen after k consecutive Insert?

Fibonacci Heap Operation

- Union(H_1,H_2):

  It puts the trees in H_1 and H_2 together, forming a new heap H

  - does **not** merge any trees into one

  Set min(H) accordingly

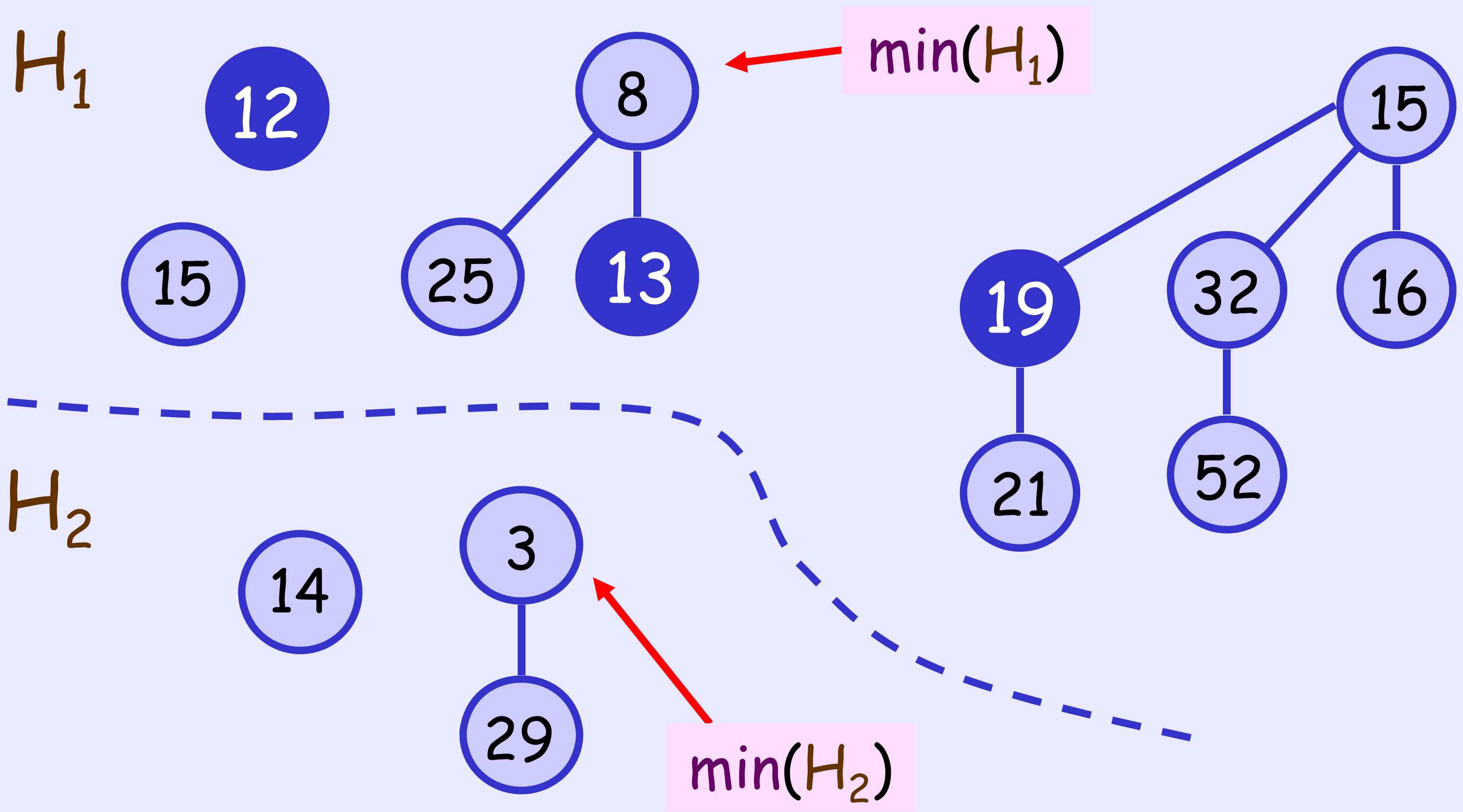
  ➔ t(H) and m(H) unchanged

  ➔ amortized cost = 2 + 0 = O(1)

Put trees together, and set min(H)

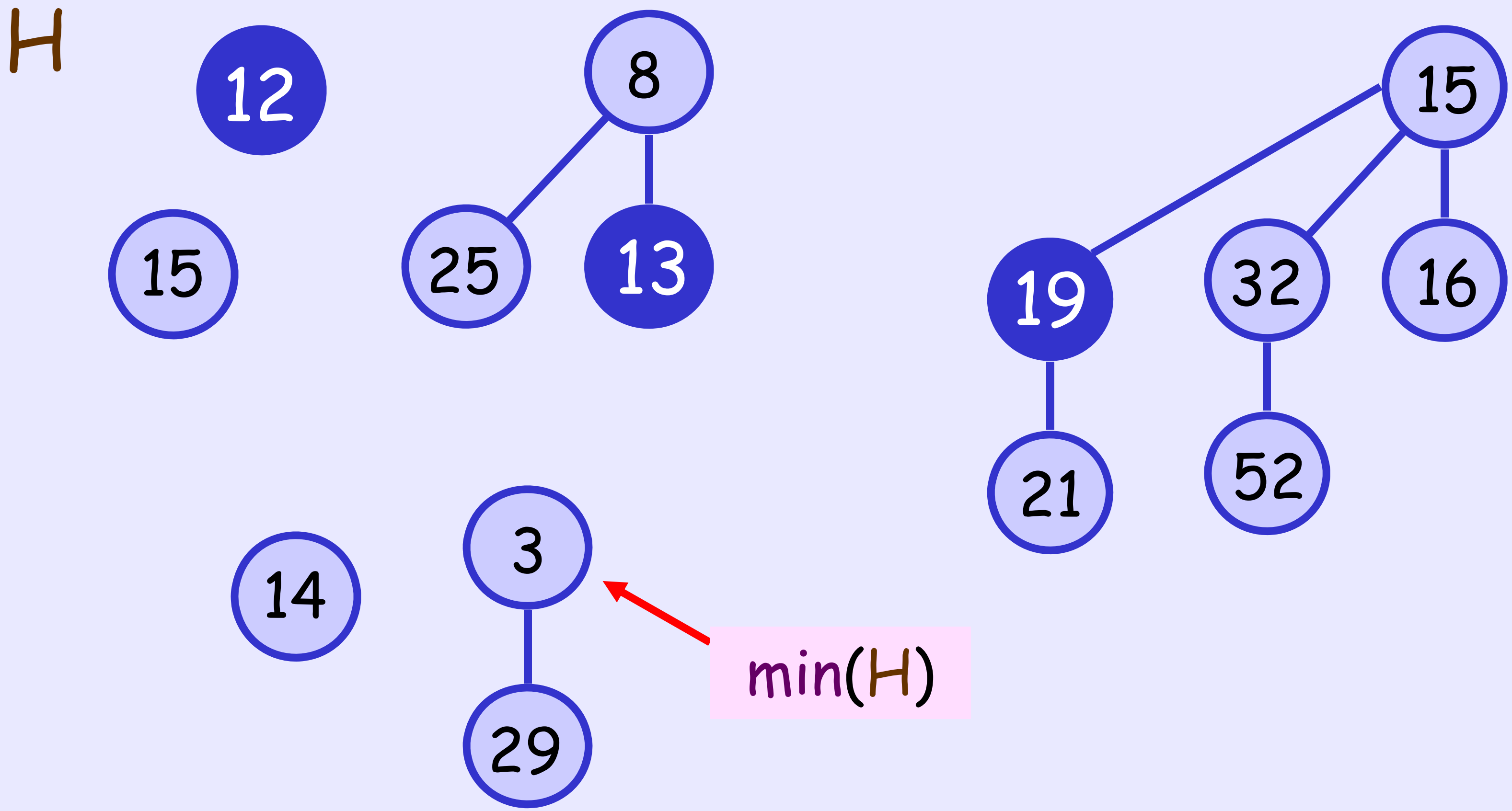
Union (Example)

Before Union



Union (Example)

After Union



Fibonacci Heap Operations

- Insert and Union are both very lazy…
- Extract-Min is a hardworking operation

➔ It reduces the #trees by joining them together

- What if Extract-Min is also lazy ??
  - a sequence of $n/2$ Insert and $n/2$ Extract-Min has worst-case $O(n^2)$ time

Extract-Min

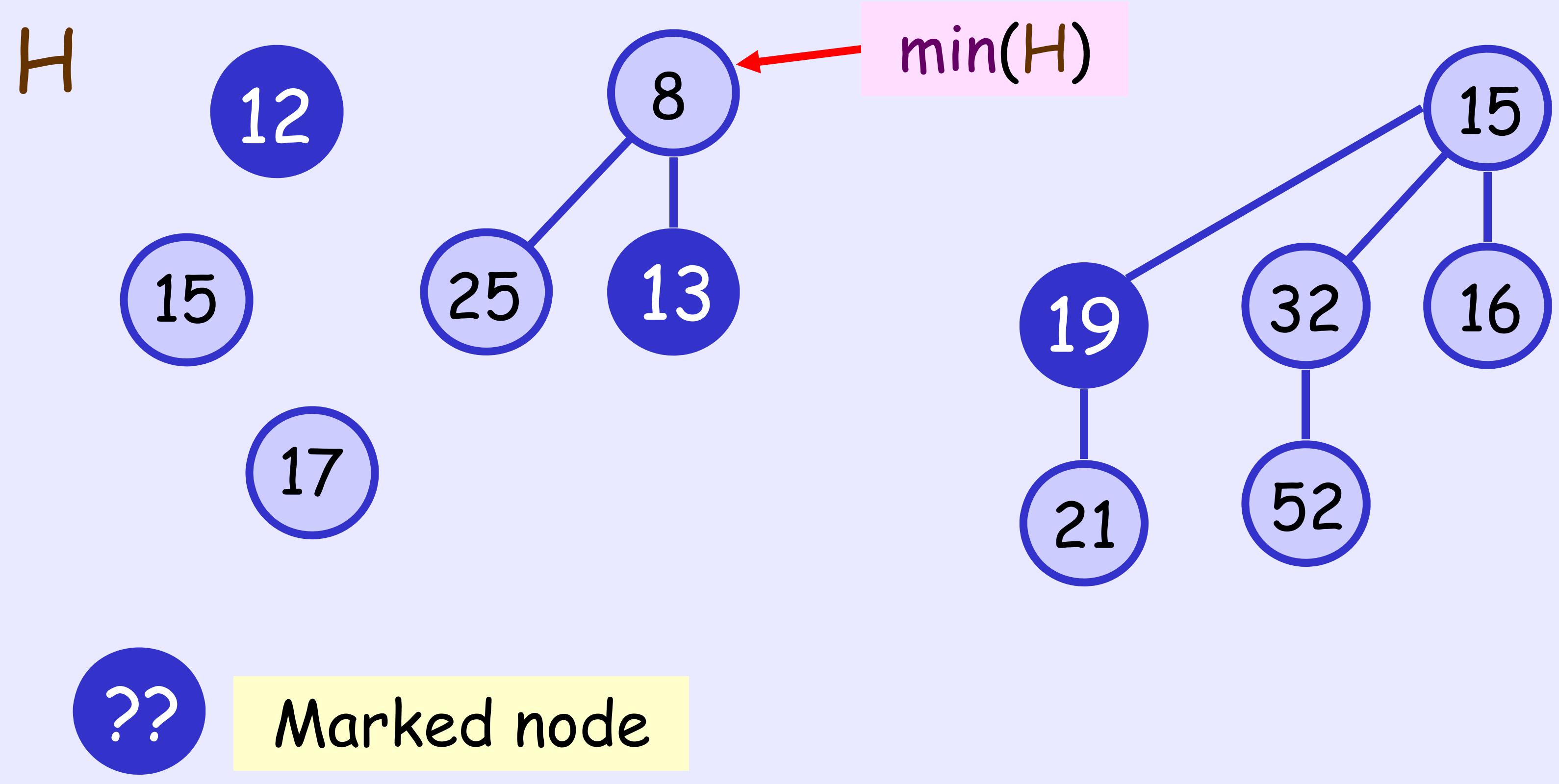
- Two major steps:
  1. Remove node with minimum key ➔ its children form roots of new trees in H
  2. Consolidation: Repeatedly joining roots of two trees with same degree

     ➔ in the end, the roots of any two trees do not have same degree

** During consolidation, if a marked node receives a parent ➔ we unmark the node

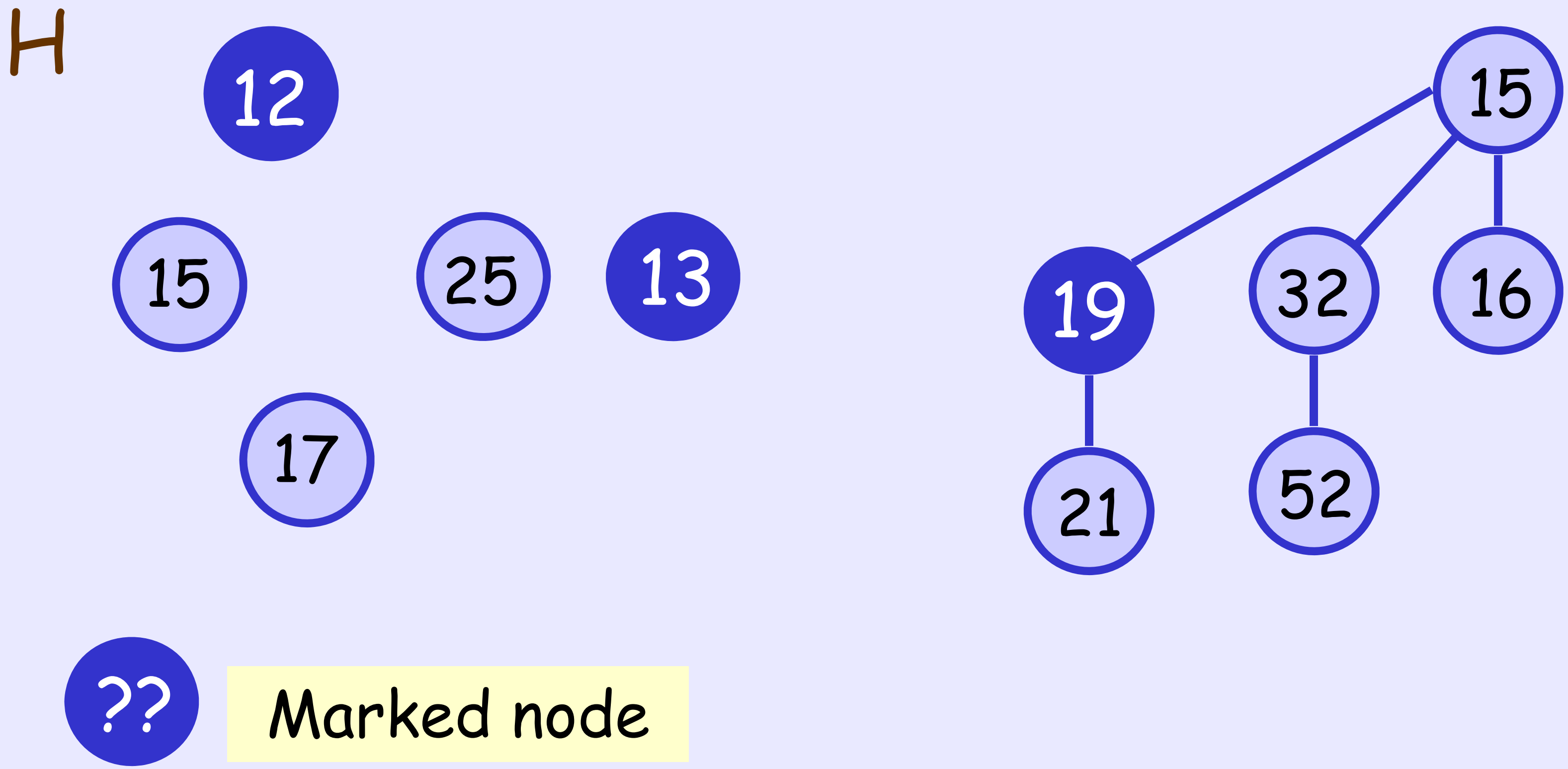
Extract-Min (Example)

Before Extract-Min



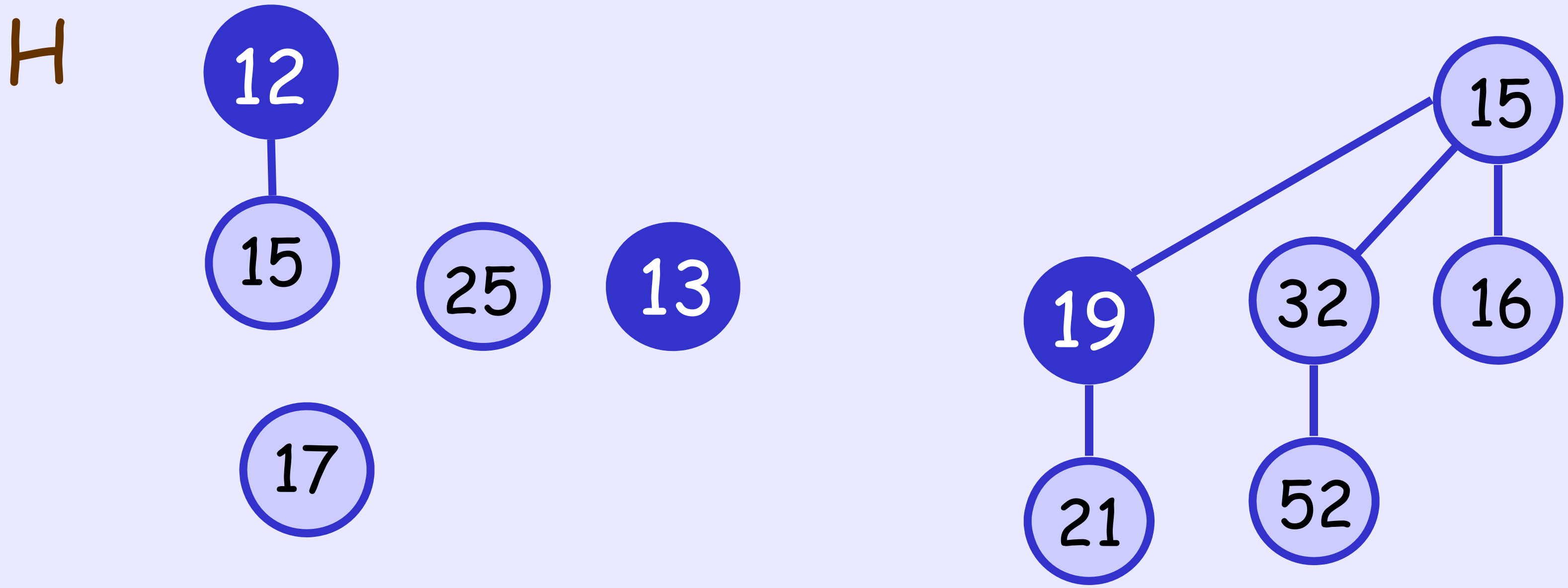
Extract-Min (Example)

Step 1: Remove node with min-key



Extract-Min (Example)

Step 2: Consolidation



Marked node

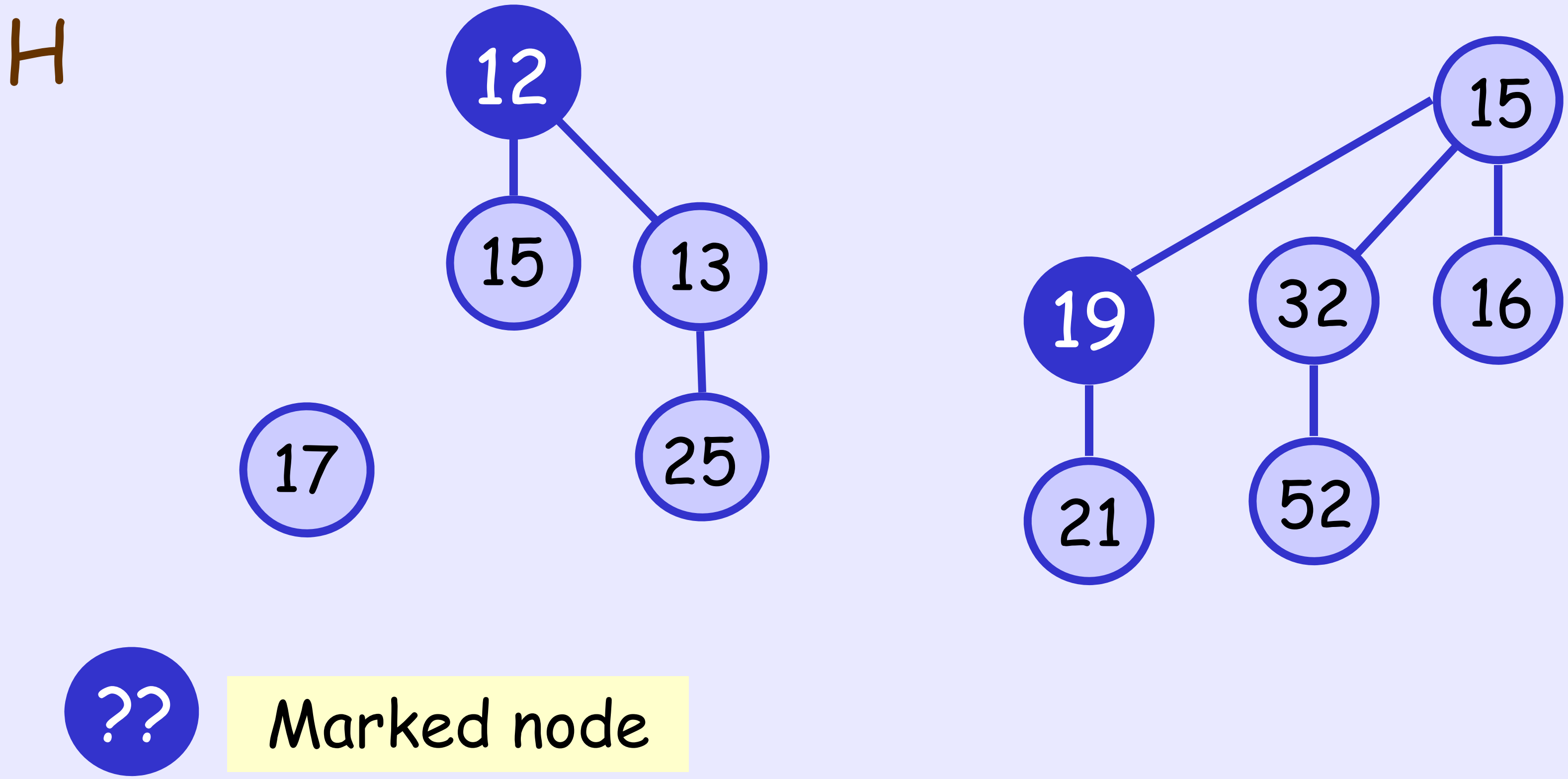
Extract-Min (Example)

Step 2: Consolidation

Marked node

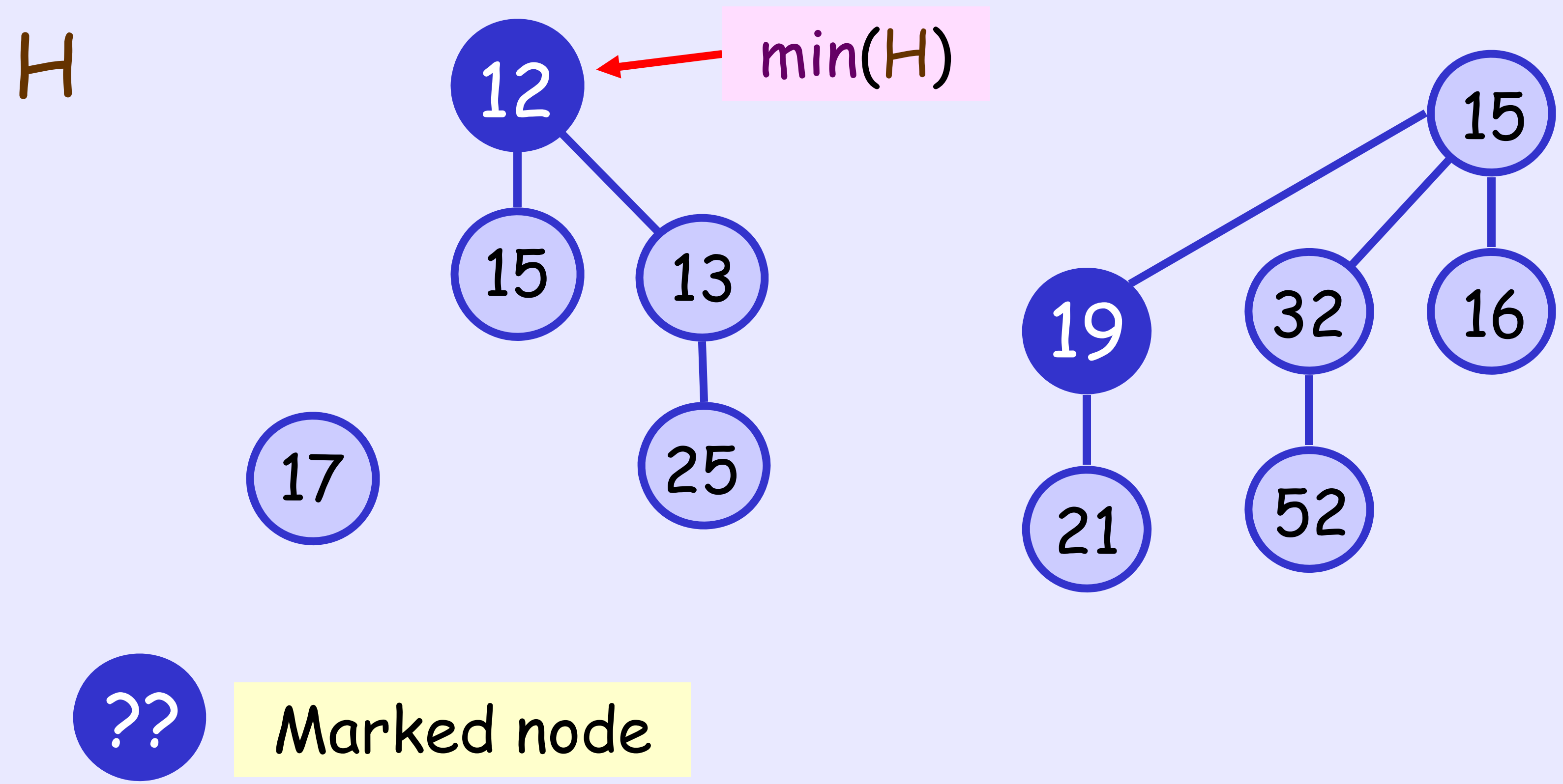
Extract-Min (Example)

Step 2: Consolidation



Extract-Min (Example)

Step 3: After consolidation, update min(H)



More on Consolidation

- The consolidation step will examine each tree in H one by one, in arbitrary order
- To facilitate the step, we use an array of size MaxDeg(n)

[Recall: MaxDeg(n) $\geq$ max deg of a node in final heap]

- At any time, we keep track of at most one tree of a particular degree

  → If there are two, we join their roots

Amortized Cost

- Let H' denote the heap just before the Extract-Min operation

→ actual cost: O(t(H') + MaxDeg(n))

potential before: t(H') + 2m(H')

potential after:

at most MaxDeg(n) + 1 + 2m(H')

[since #trees ≤ MaxDeg(n) +1, and no new marked nodes]

→ amortized cost ≤ 2MaxDeg(n) + 1 = O(log n)