04-630
Data Structures and Algorithms for Engineers

Lecture 21: Advanced Data Structures: *B-Trees & Fibonacci Heaps*

George Okeyo (with Modifications by Theophilus Benson)

Carnegie Mellon University Africa

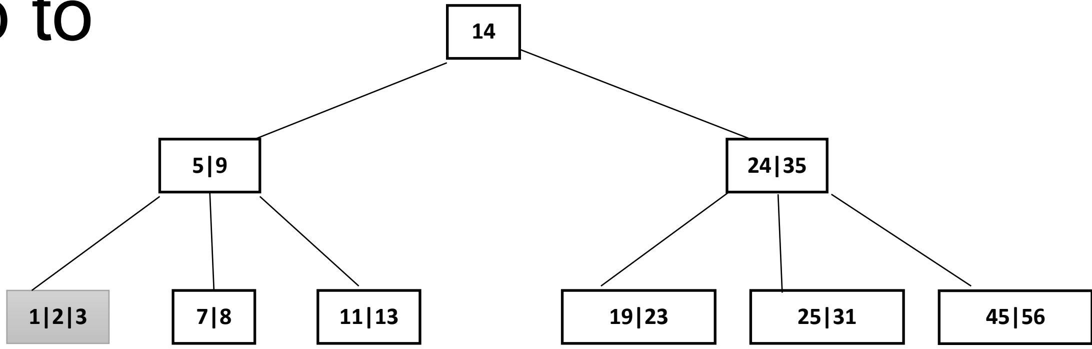
gokeyo@andrew.cmu.edu

Agenda

- B-trees and applications
- Fibonacci heaps and applications

B-tree: Overview

- ***Definition***: A B-tree is an *m-way tree*(i.e., a tree where each node may have up to m children) and:
    - The number m should always be *odd*.
    - The B-tree is said to be of *order m=2t-1*.
- A *large branching factor* reduces the height of the tree

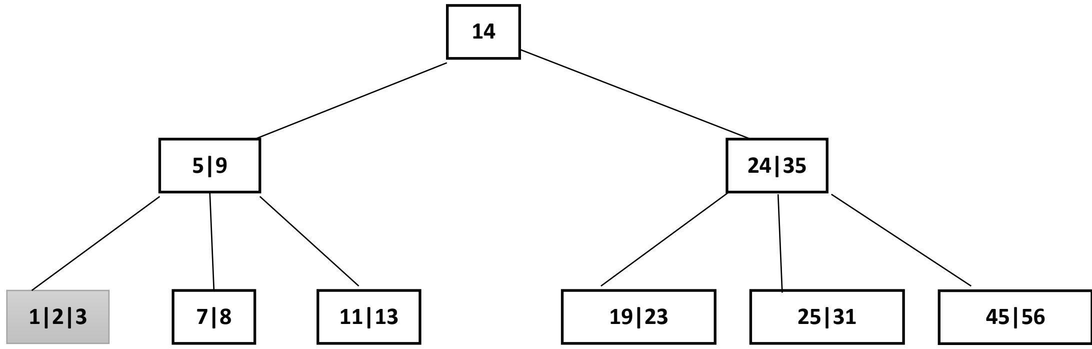


Practical Applications: B-trees

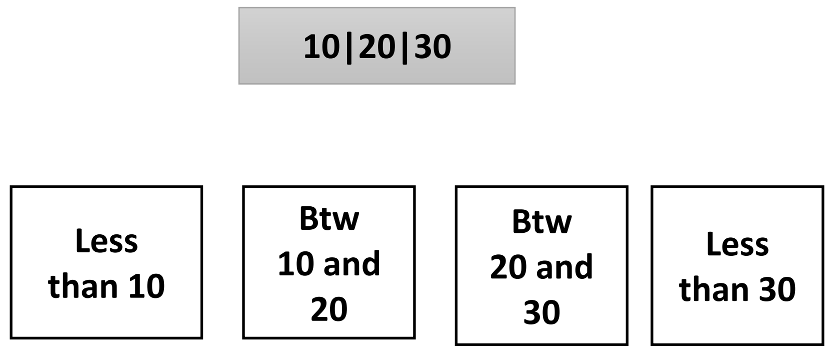
- Accessing data from secondary memory
  - Often the access time for a page may be large but the time to read the information once accessed is small.
  - A B-tree can be applied to efficiently access secondary storage by setting the size of a node to the size of the page.
  - The large branching factor reduces the number of disk accesses needed to find any page.
  - The number of disk accesses required on a B-tree is proportional to the tree’s height.
  - A B-tree with a branching factor of 1001 and height 2 stores over one billion pages and since the root is kept permanently in memory only two disk accesses are required to find a page.

B-trees: overview

- B-trees are *balanced search trees*.
- Nodes can have *multiple children*.
- Every leaf node has the same depth- *the tree's height*.
- There is an *upper and a lower bound* on the number of elements a leaf can hold.



B-trees: overview(2)



- There is an *upper and a lower bound* on the number of elements a leaf can hold.
- Bounds can be expressed in terms of a fixed integer t.
  - Every node, other than the root must have at least *t-1 keys (~m/2 -1).*
  - An internal node has at least *t children (~m/2)*.
  - Every node can have at most *2t-1 children. (or m)*.
  - The maximum number of keys per node are *m-1 or 2(t-1); m is the order of the tree.*

B-tree operations: insertion

- Attempt to insert the ***new key into a leaf***
- If this results in the leaf being larger than m-1, split the leaf into two, ***promoting the middle key to the leaf's parent***.
- If as a result the parent becomes too big (greater than m-1), split the parent into two, ***promoting the middle key***.

B-tree operations: insertion

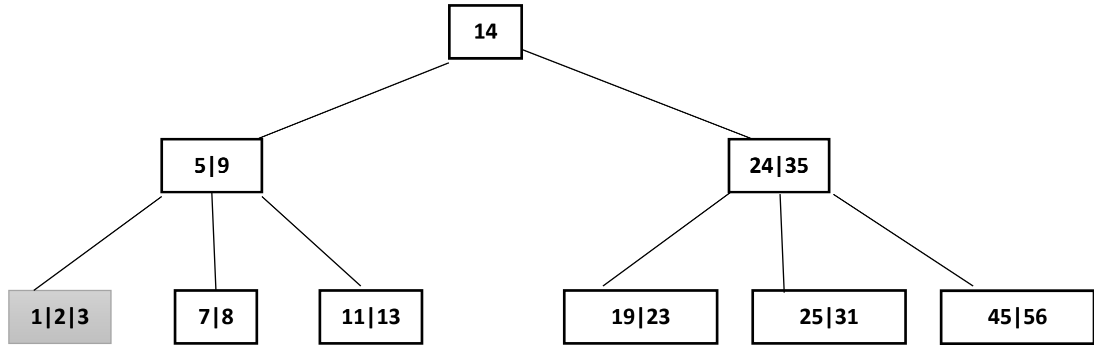
- Repeat the strategy until ***B-tree conditions*** are maintained.

- If the root becomes too big too as a result, ***split the root into two and make the middle key the new root***.
  - This increases the tree height.

Insert!

- Order M = 4
- Max keys in a node = 3 = (M-1)
- Min keys in a node = **[m/2]-1** =1
- Each internal node at-least m/2 children = 2
- A non leaf node with **n-1** keys must have **n** number of children.
- Note: ascending order

Insert 4



insertion steps:

- Start at leaf
- Promote when # keys > m-1
- Promote middle and split node

B-tree operations: deletion

- We wish to delete from the leaf. Three cases exist:

  1) If the *key is already in the leaf node and removing it does not cause the leaf to have too few keys* **(#keys>=t-1 maintained)**, simply remove the key.

  2) If the key is *not in the leaf, then it is guaranteed that its predecessor or successor* will be in a leaf.

     - Delete the key and promote the predecessor or successor key to the non-leaf deleted key's position.
     - First try predecessor then successor

B-tree operations: deletion

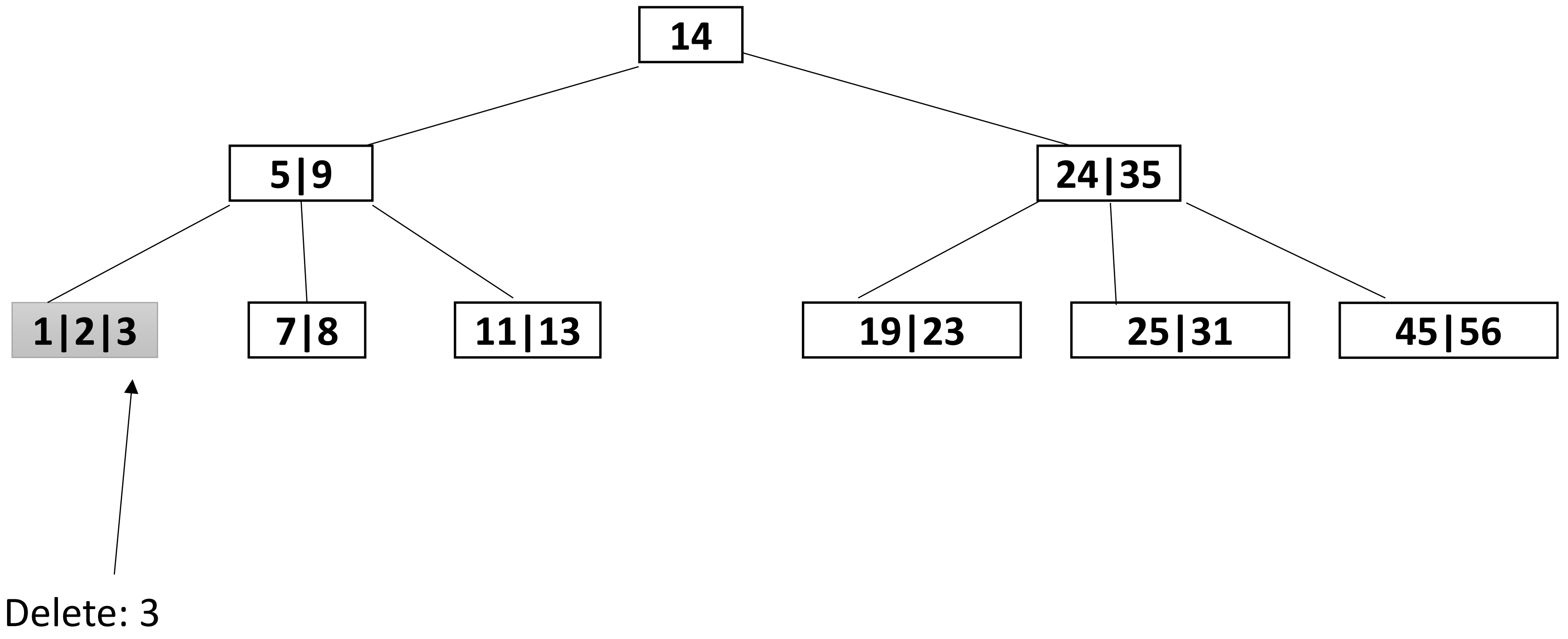
3) If the operation in (1) or (2) cause the leaf to have less than the minimum number of keys, then we look at the sibling immediately adjacent to the leaf in question.

   a) If *one of them has more than the minimum number of keys*, then we promote one of its keys to the parent and take the parent key into the deficient leaf.

   b) If *neither of them has more than the minimum number of keys then the deficient leaf and one of its neighbours can be combined with their shared parent* (the opposite of promoting a key) and the new leaf will have the correct number of keys; if this step leaves the parent with too few keys, then we repeat the process up to the root itself, if required.

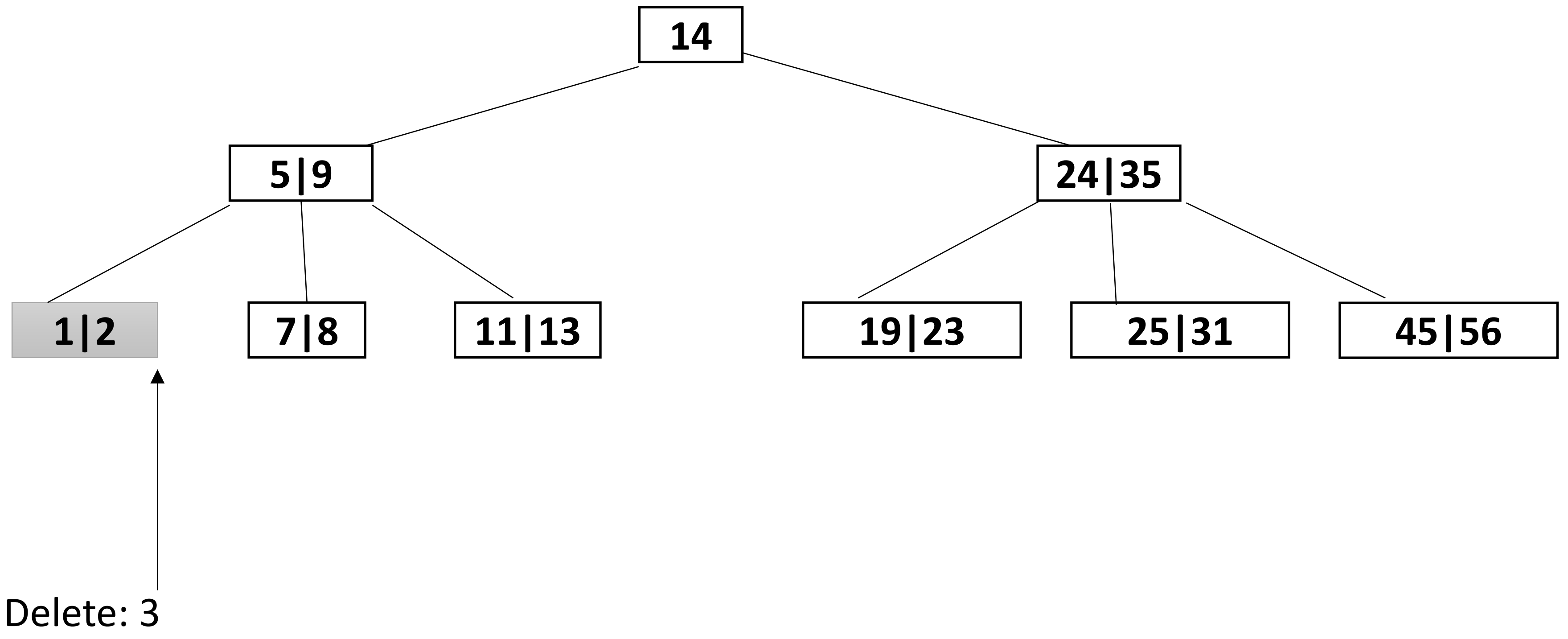
B-tree operations: deletion-Case 1 (delete from leaf).

5-way tree, m=5, t=3.



B-tree operations: deletion-Case 1 (delete from leaf)..

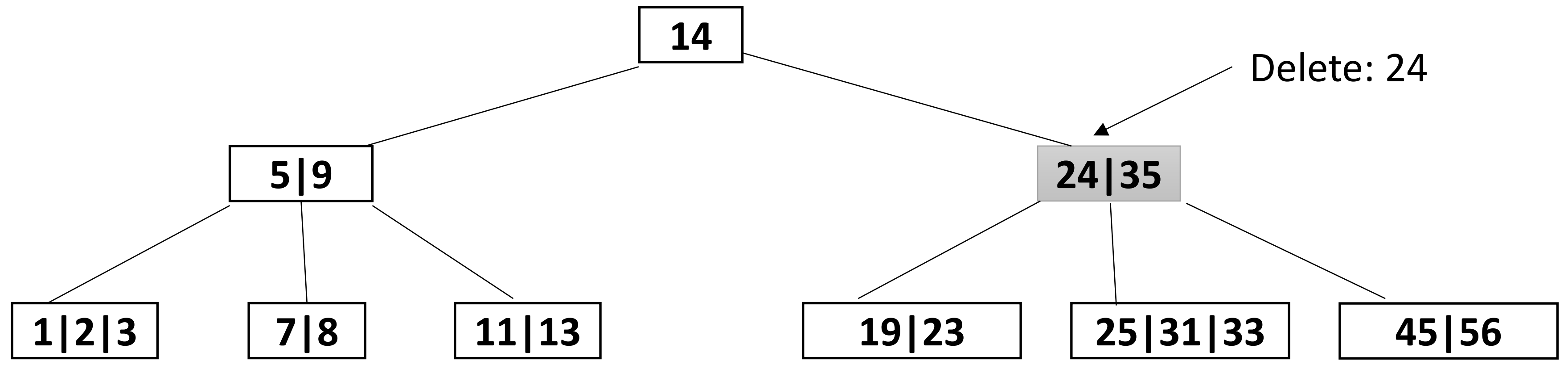
5-way tree, m=5, t=3.



There are enough keys (>=t-1, i.e. >=2), therefore just delete it.

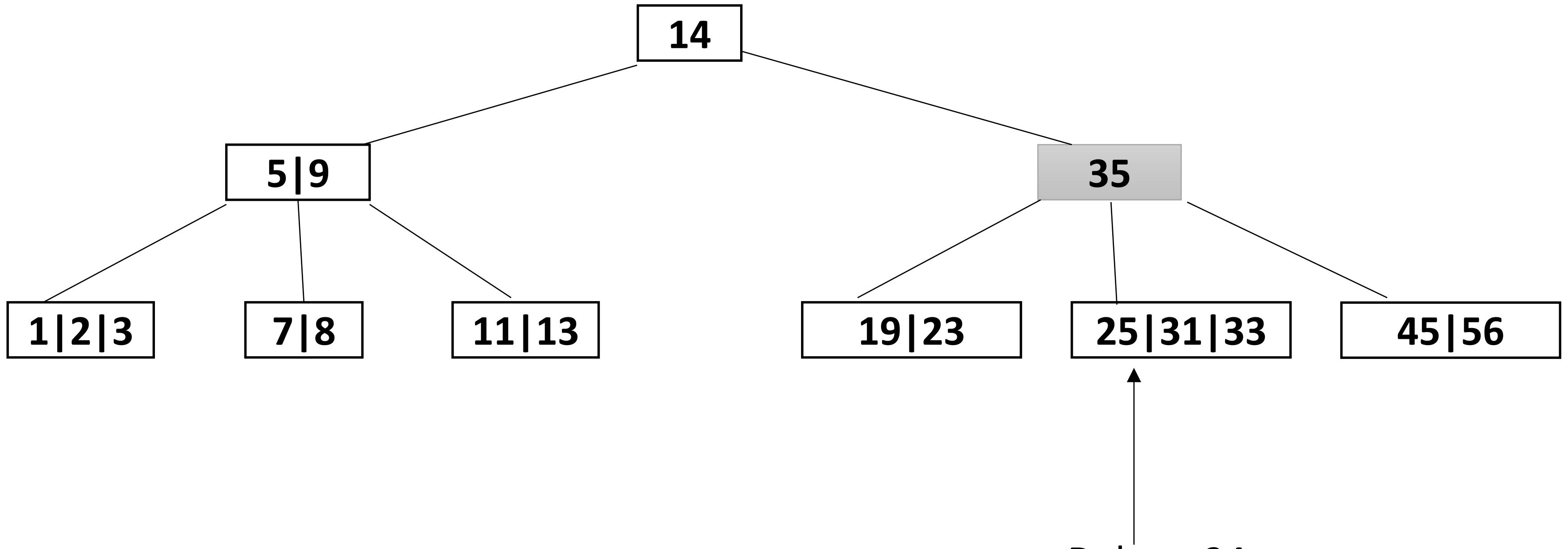
B-tree operations: deletion-Case 2 (internal node).

5-way tree, m=5, t=3.



B-tree operations: deletion-Case 2 (internal node).

5-way tree, m=5, t=3.

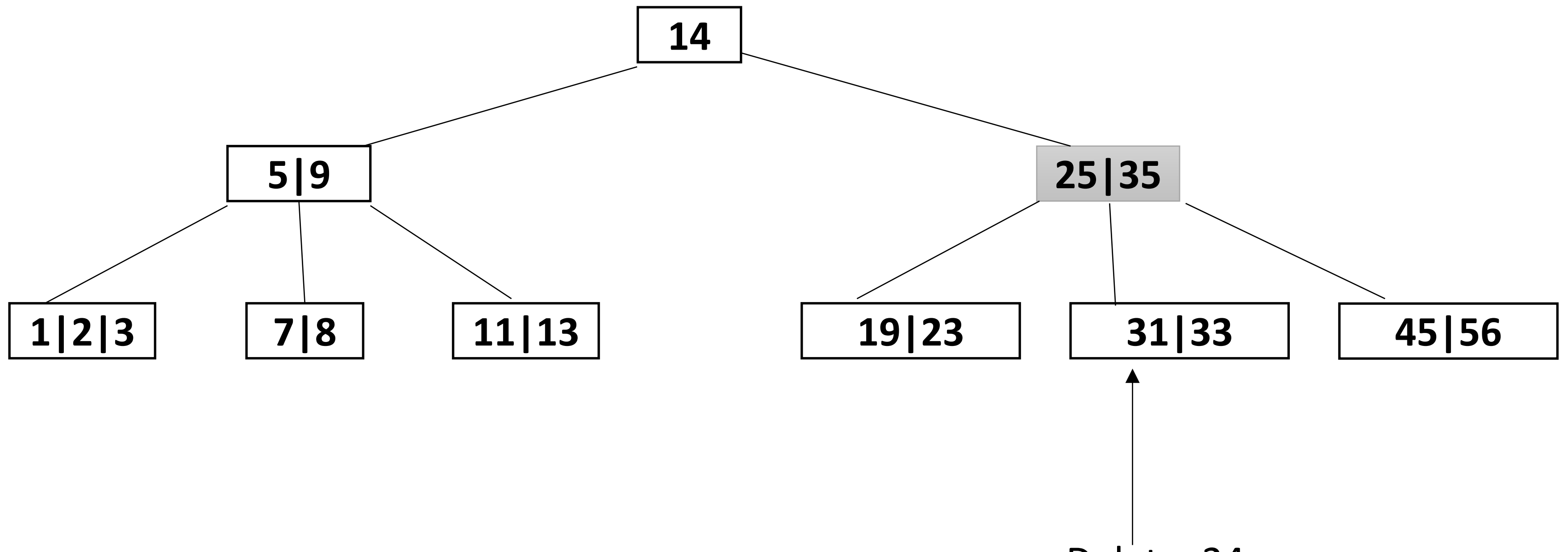


Delete: 24

Borrow successor or predecessor. In this case, successor.

B-tree operations: deletion-Case 2 (internal node).

5-way tree, m=5, t=3.

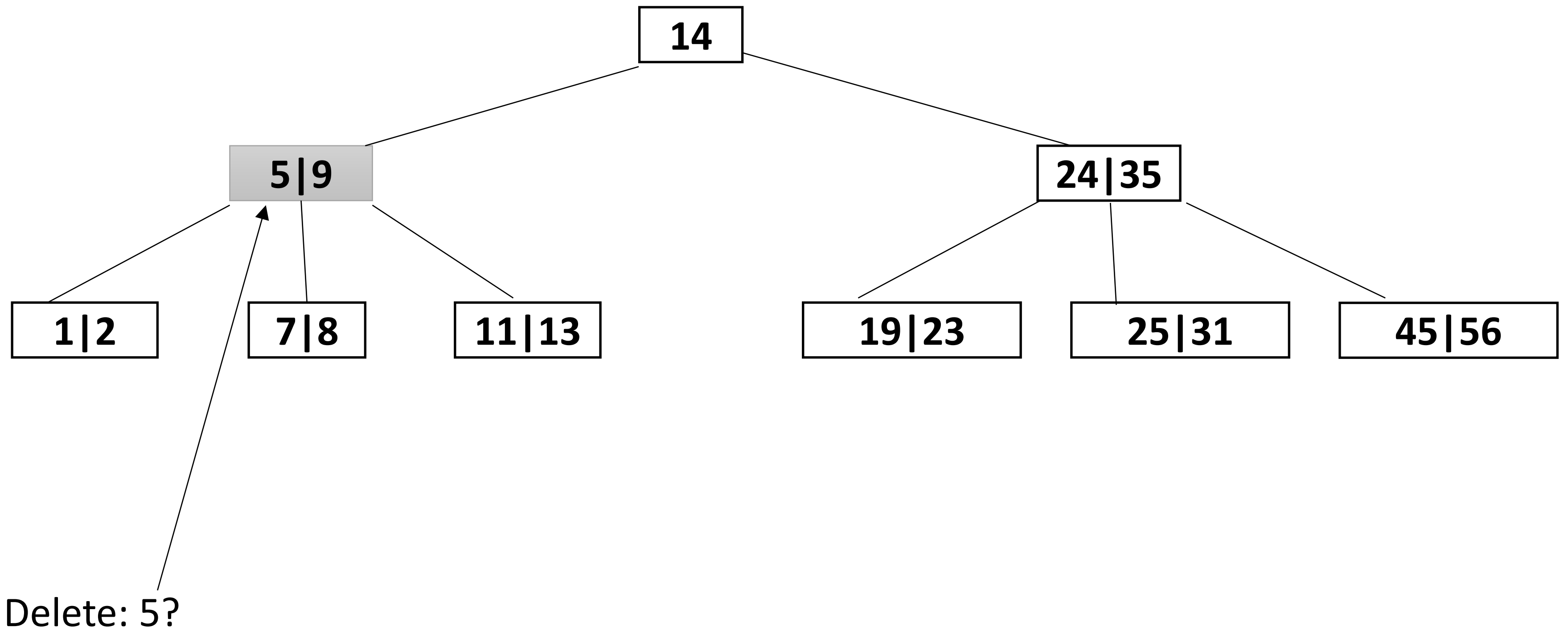


Delete: 24

Borrow successor or predecessor. In this case, successor.

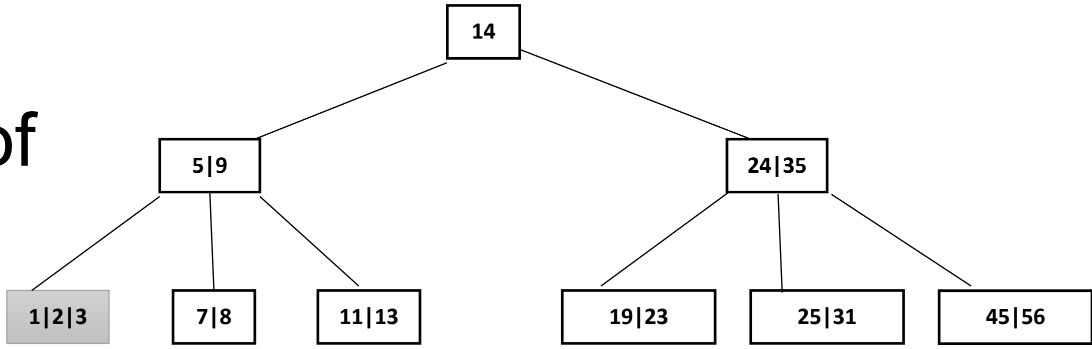
B-tree operations: deletion-Case 3.

5-way tree, m=5, t=3.



Complexity Analysis

- Searching a B-tree is similar to searching a BST except that a multi-way branching decision is made at each node according to the number of the node's children.
  - Time: O(t $\log_t$ n)
  - This is the same for **insertion** and **deletion.**
- Splitting a node takes O(t $\log_t$ n)



O(t $\log_t$ n)

Linear scan of keys
In each node

Height of tree

2-3 Trees

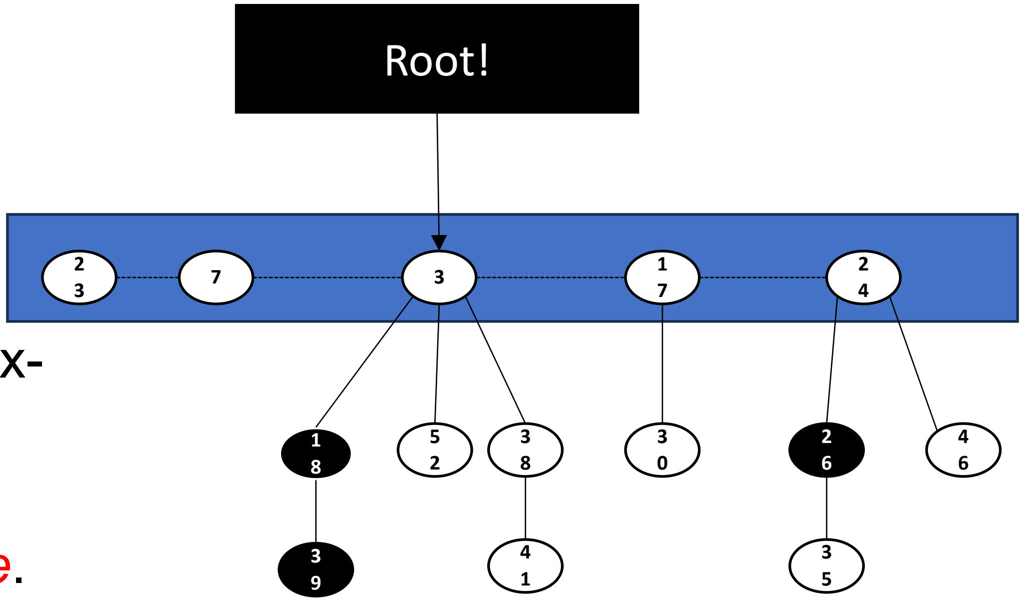
- A B-tree of order m=3 is called a 2-3 Tree (2 keys, 3 children/branches)

- Each internal node has either 2 or 3 children.

Fibonacci Heaps

Why do we care? Shortest path and Minimum spanning Tree!!!!

Fibonacci Heaps: overview



- Fibonacci heap:
  - A *list* of *min-heap ordered trees*(can also be max-heap).
  - Supports a set of operations that constitute a "*mergeable heap*".
  - Several operations run in *constant amortized time*.
  - **Delay work as much as possible!!!!**

- Representation:
  - Uses *left-child*, *right sibling pointers* and *circular doubly-linked list*.
  - *Connect roots* of trees with a circular doubly linked list called the ***root list***.
  - Provide a *pointer to the root* of the *tree with the min element*.

- Binomial heap: eagerly consolidate trees after each *insert*.

- Fibonacci heap: lazily defer consolidation until next *delete-min*.

Fibonacci Heaps: overview(2)

- Why circular, doubly-linked list?

- Two advantages:

  - We can *remove a node in O(1) time*.

  - Given two such lists, we can *concatenate them in O(1) time*.

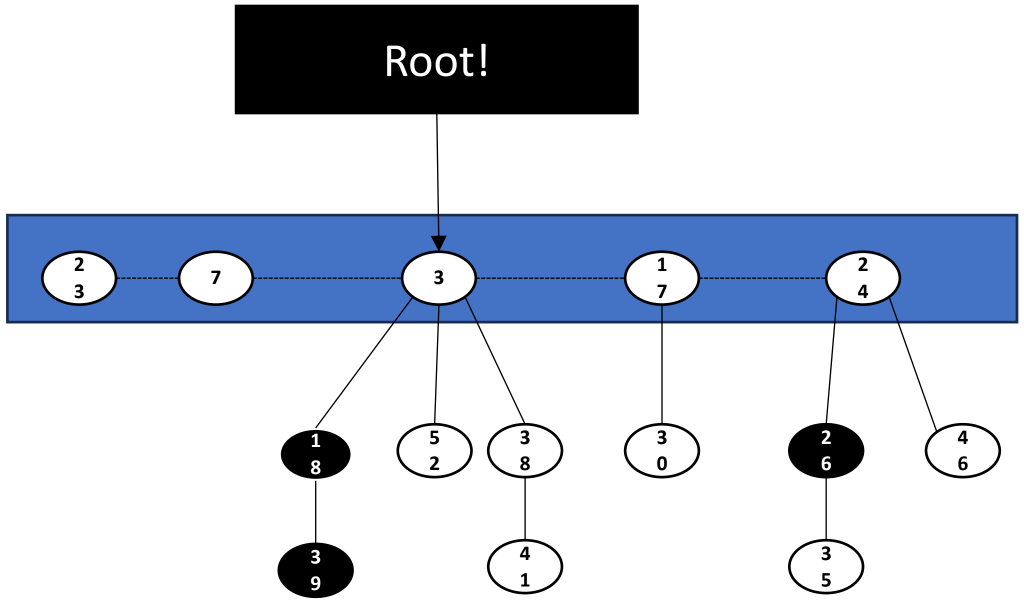
Fibonacci Heaps: overview (3)

Important information:

- Degree(x)- degree of node x. The number of children in the child list of x.
- Mark(x)- mark of node x (black or gray). A Boolean-valued field.
    - Indicates if node x has lost a child since the last time x was made the child of another one.
    - *Newly created* nodes are *unmarked*.
    - A node x becomes *unmarked whenever it is made a child of another node*.
- t(H)- number of trees
- m(H)- number of marked nodes.
- $\Phi(H)=t(H)+2\,m(H)$- potential function
    - A Fibonacci heap begins with an empty heap: $\Phi(H)=0$

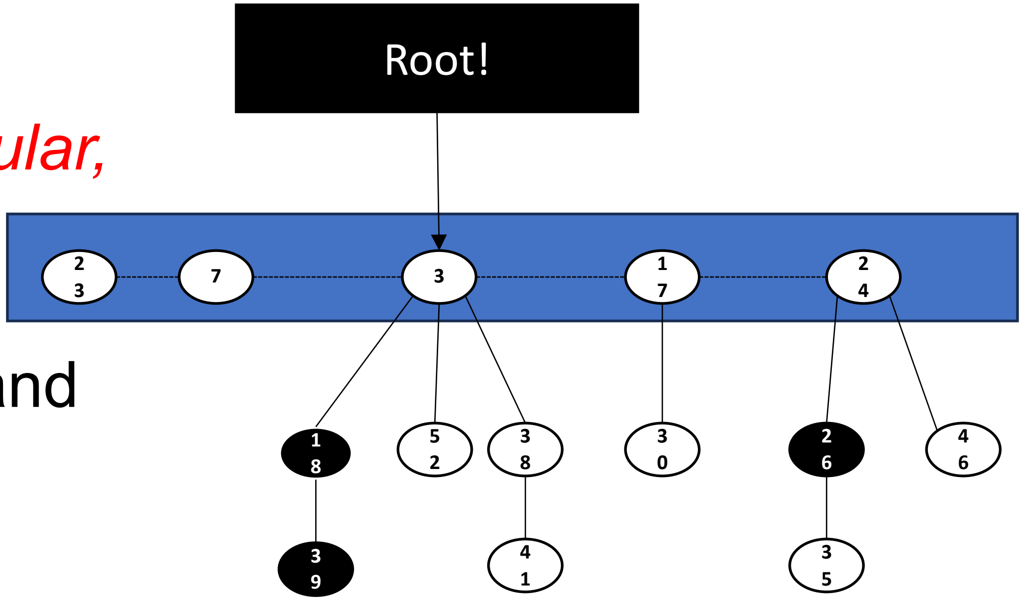
Fibonacci heaps: implementation

- We represent each element by a *node* within a tree, and each node has a *key attribute*.
- The rooted trees are *min-heap(max-heap) ordered*.
- Each tree obeys *min-heap(max-heap) property*.
- Each node x, contains a pointer x.p to its parent and pointer x.child to as any of its children.

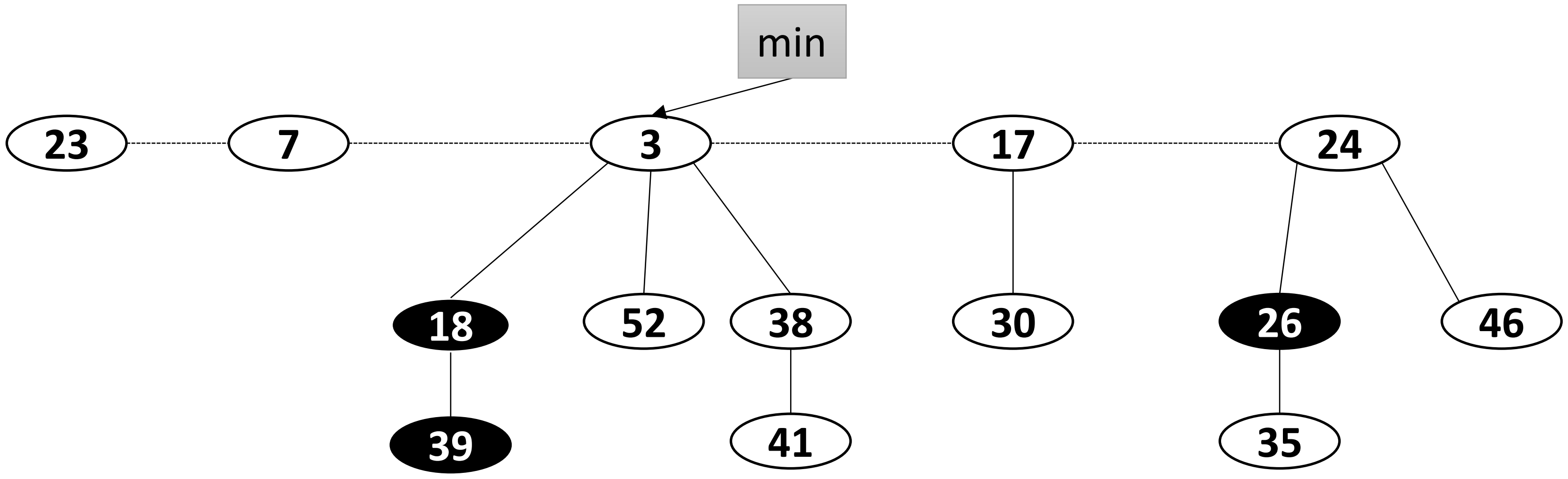


Fibonacci heaps: implementation(2)

- The children of x are linked together in a *circular, doubly linked list*, i.e. the child list of x.
- Each child y in a child list has pointers y.left and y.right that point to y's left and right siblings, respectively.
- If node y is an only child, then y.left=y.right=y.
- Siblings may appear in a child list in any order.



Fibonacci heap: implementation



5 min-heap-ordered trees.

14 nodes.

3 black nodes marked.

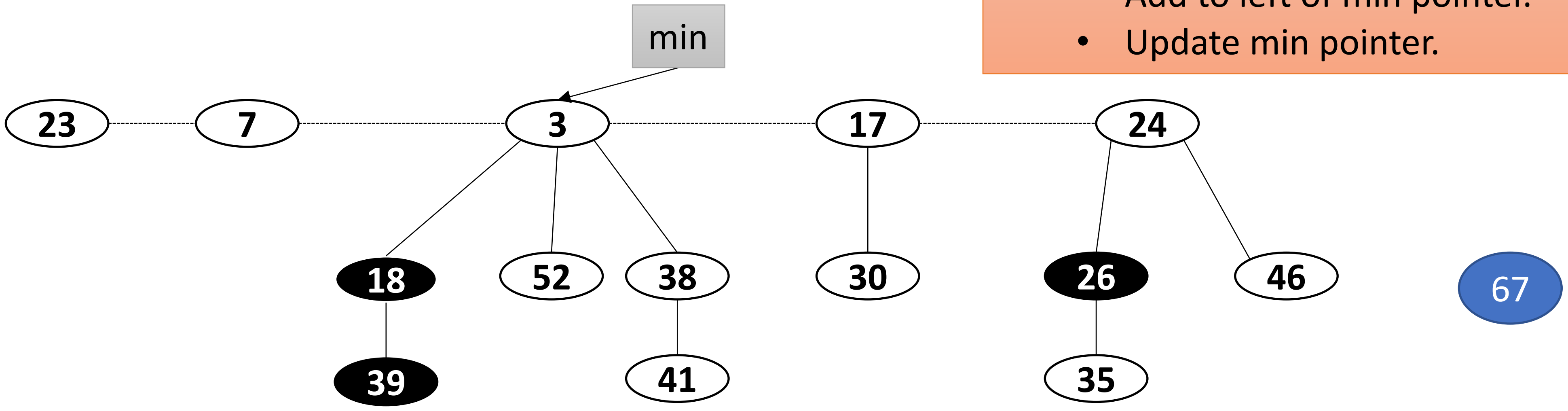
Mergeable heaps

- Refers to any data structure that supports the following five operations(*mergeable-heap operations*):
  - Make-Heap : creates an empty new heap.
  - Insert(H,x): inserts element x, whose key has already been filled in, into the heap H.
  - Minimum(H): return a pointer to the element whose key is minimum.
  - Extract-Min(H): deletes and returns the element whose key is minimum.
  - Union(H1,H2): returns a new heap(made up of all elements of H1 and H2) and destroys H1 and H2.

Fibonacci heap: insert

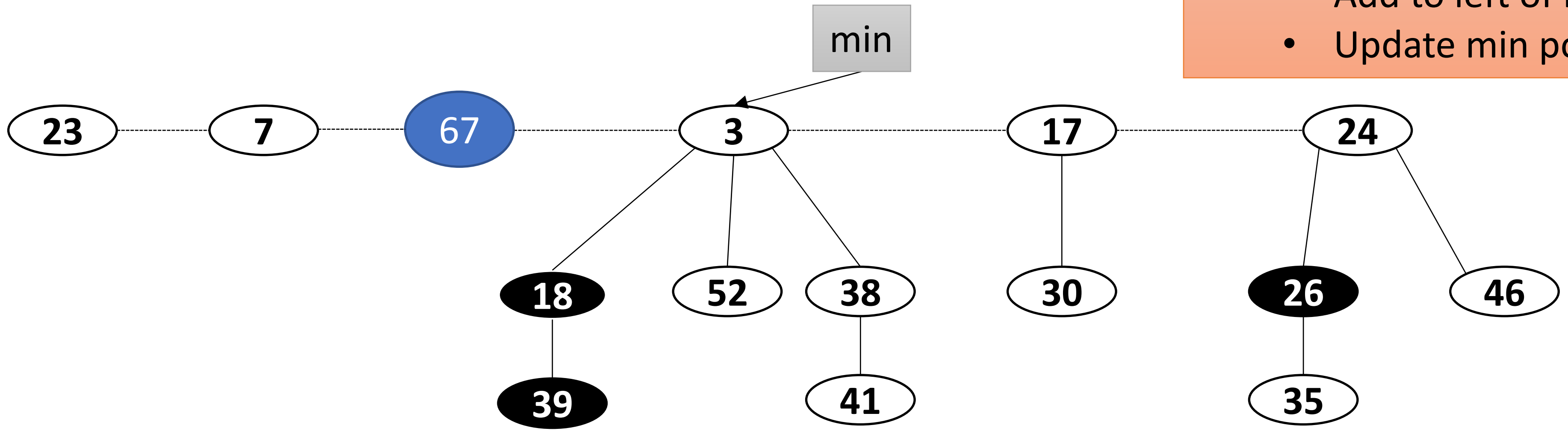
Insert: 67

- Create a new singleton tree.
- Add to left of min pointer.
- Update min pointer.



Fibonacci heap: insert

Insert: 67
- Create a new singleton tree.
- Add to left of min pointer.
- Update min pointer.



Complexity Analysis

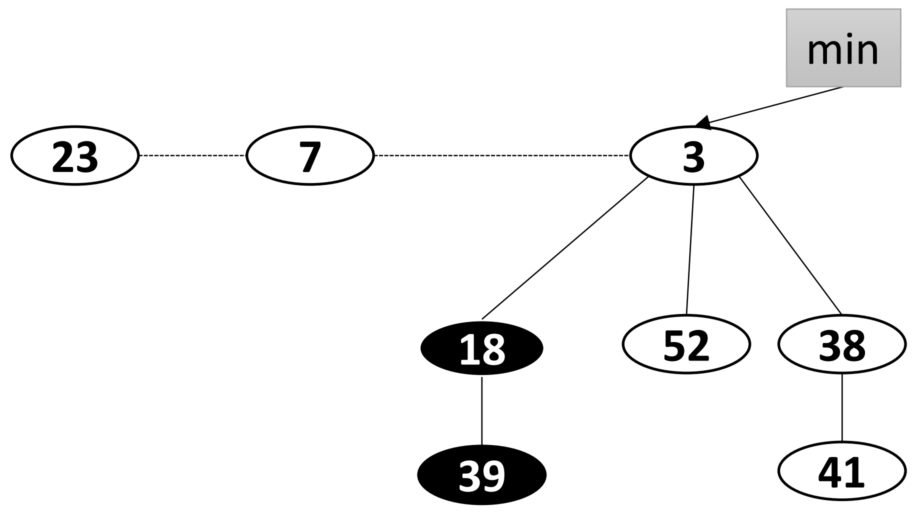
Operation	Binary Heap (worst case)	Fibonacci Heap (amortized)
Make-Heap	O(1)	O(1)
Insert	O(log n)	O(1)
Minimum	O(1)	O(1)
Extract-Min	O(log n)	O(log n)
Union	O(n)	O(1)
Decrease-Key	O(log n)	O(1)
Delete	O(log n)	O(log n)

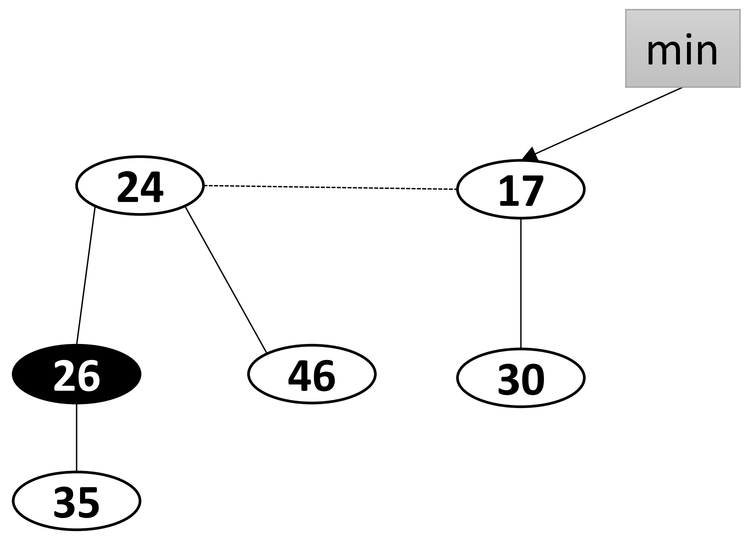
Note: Fibonacci Heaps are desirable when the number of ***Extract-Min*** and ***Delete*** operations is small relative to the number of other operations.

Mergeable heaps

- Refers to any data structure that supports the following five operations(*mergeable-heap operations*):
  - Make-Heap : creates an empty new heap.
  - Insert(H,x): inserts element x, whose key has already been filled in, into the heap H.
  - Minimum(H): return a pointer to the element whose key is minimum.
  - Extract-Min(H): deletes and returns the element whose key is minimum.
  - Union(H1,H2): returns a new heap(made up of all elements of H1 and H2) and destroys H1 and H2.

Fibonacci heap: union

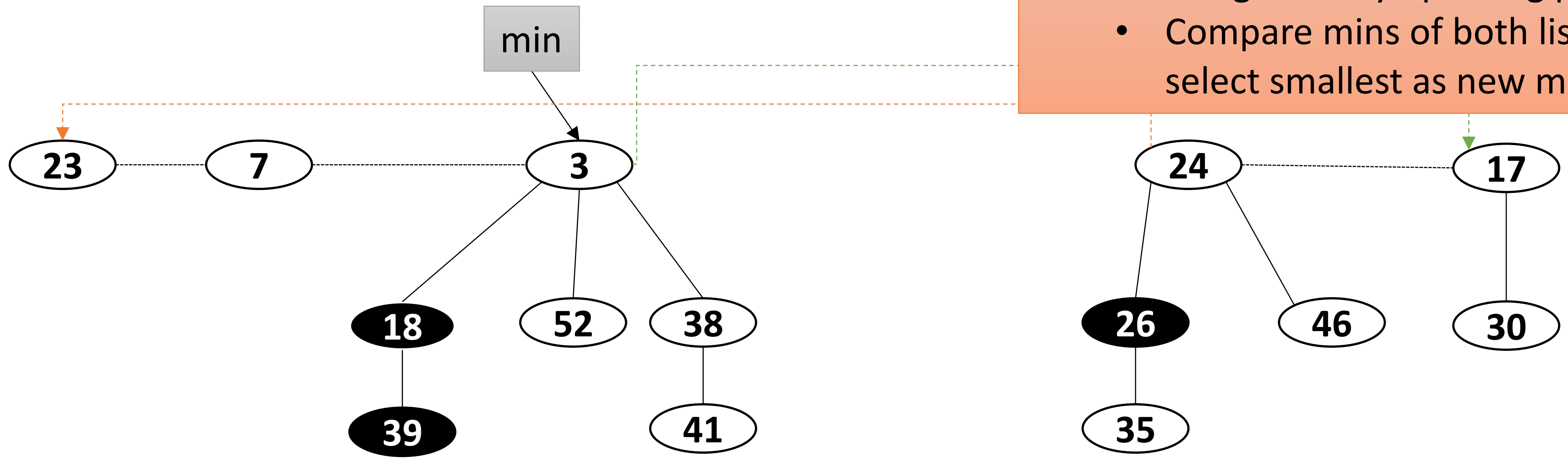




Fibonacci heap: union

Union

- Merge lists by updating pointers
- Compare mins of both list and select smallest as new min



Complexity Analysis

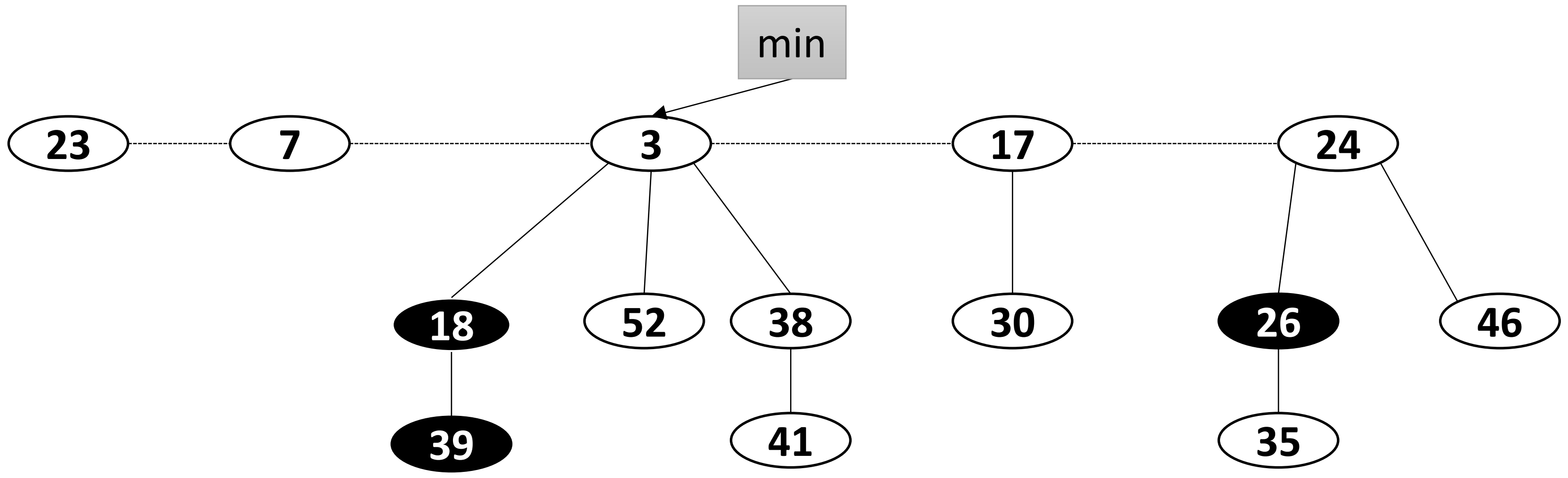
Operation	Binary Heap (worst case)	Fibonacci Heap (amortized)
Make-Heap	O(1)	O(1)
Insert	O(log n)	O(1)
Minimum	O(1)	O(1)
Extract-Min	O(log n)	O(log n)
Union	O(n)	O(1)
Decrease-Key	O(log n)	O(1)
Delete	O(log n)	O(log n)

Note: Fibonacci Heaps are desirable when the number of ***Extract-Min*** and ***Delete*** operations is small relative to the number of other operations.

Mergeable heaps

- Refers to any data structure that supports the following five operations(*mergeable-heap operations*):
  - Make-Heap : creates an empty new heap.
  - Insert(H,x): inserts element x, whose key has already been filled in, into the heap H.
  - Minimum(H): return a pointer to the element whose key is minimum.
  - Extract-Min(H): deletes and returns the element whose key is minimum.
  - Union(H1,H2): returns a new heap(made up of all elements of H1 and H2) and destroys H1 and H2.

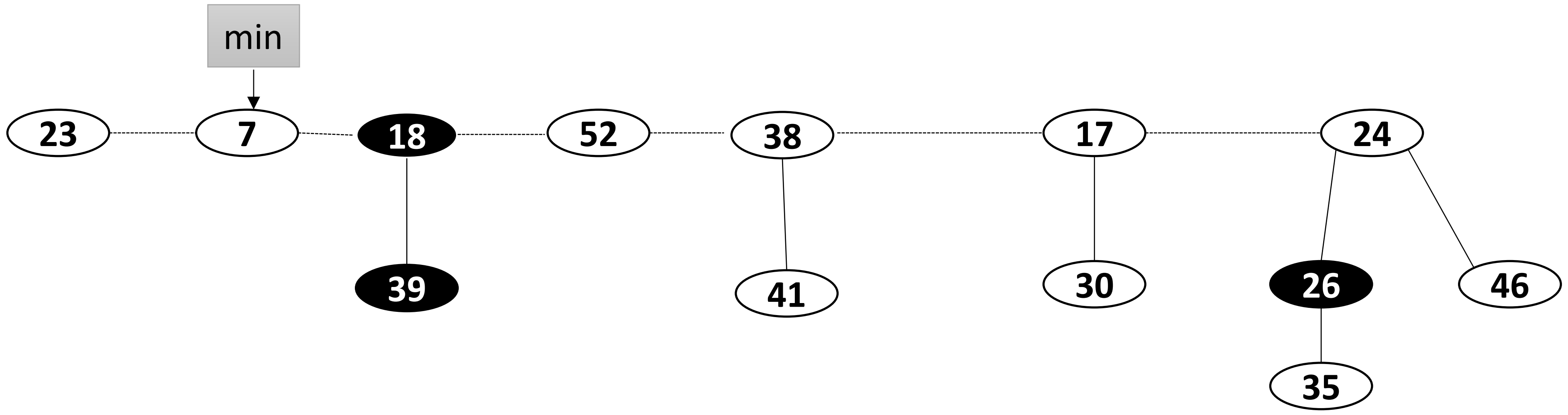
Fibonacci heap: delete min

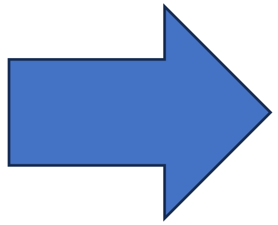


Steps:

1) Delete min and concatenates the children to the root list.
2) Consolidate the trees so that no two roots have the same degree.

Fibonacci heap: delete min

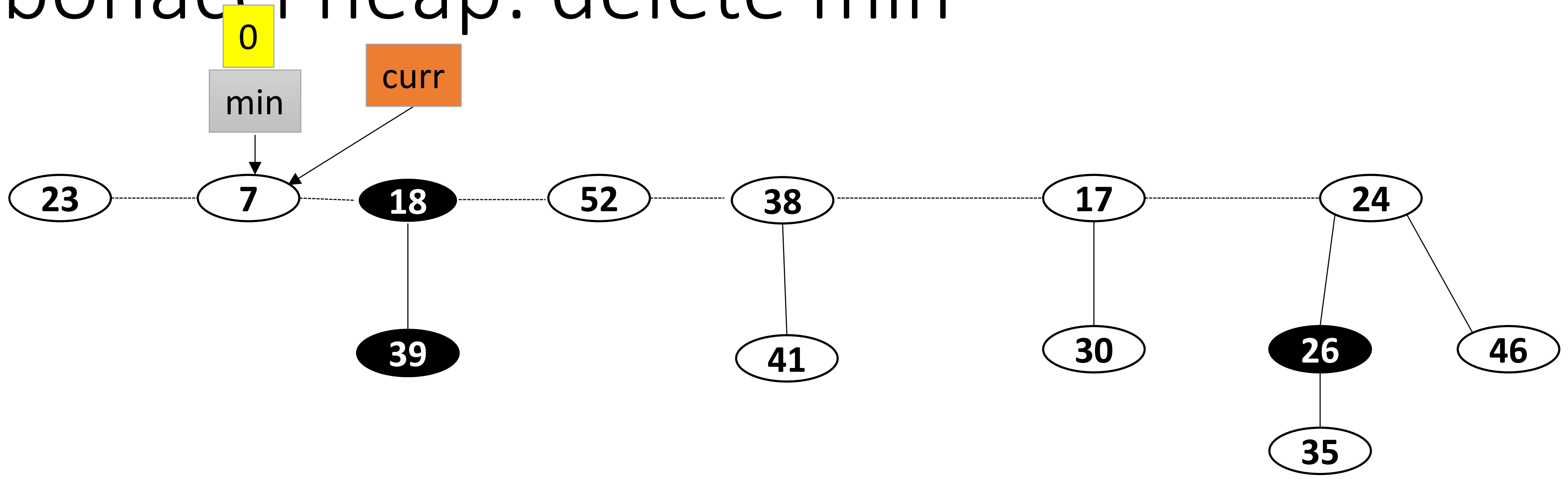


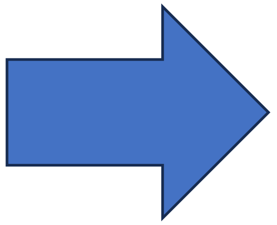


Steps:

1) Delete min and concatenates the children to the root list.
2) Consolidate the trees so that no two roots have the same degree.

Fibonacci heap: delete min

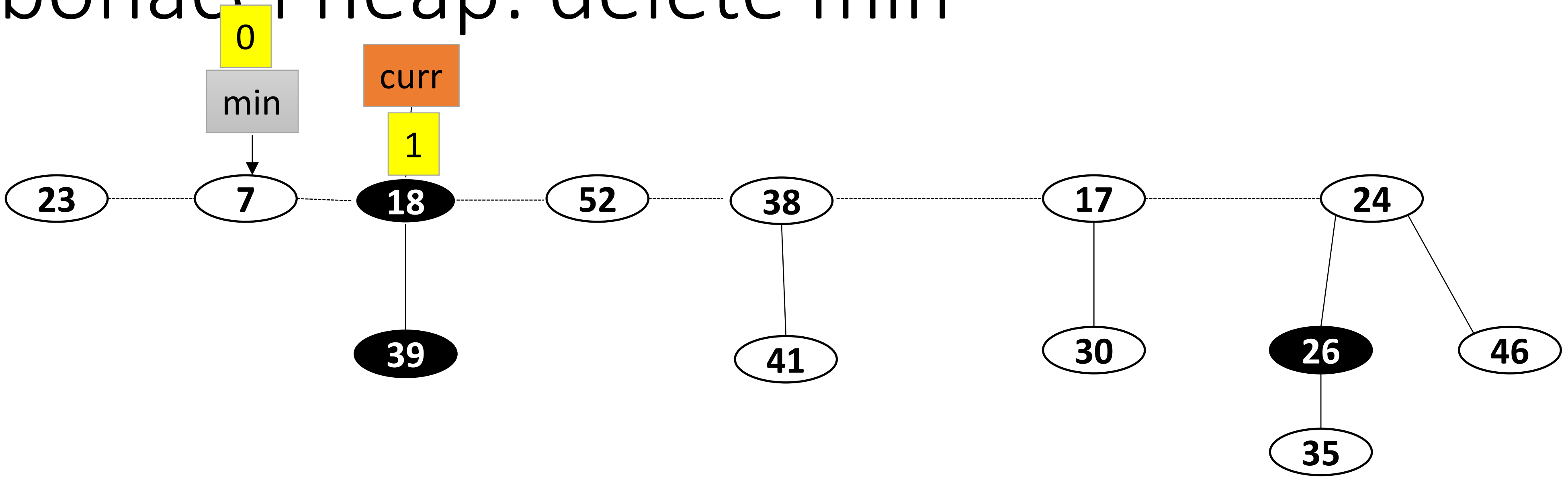


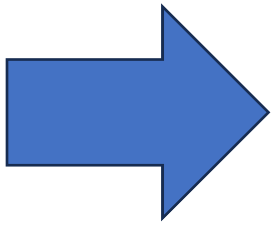


Steps:

1) Delete min and concatenates the children to the root list.
2) Consolidate the trees so that no two roots have the same degree.

Fibonacci heap: delete min

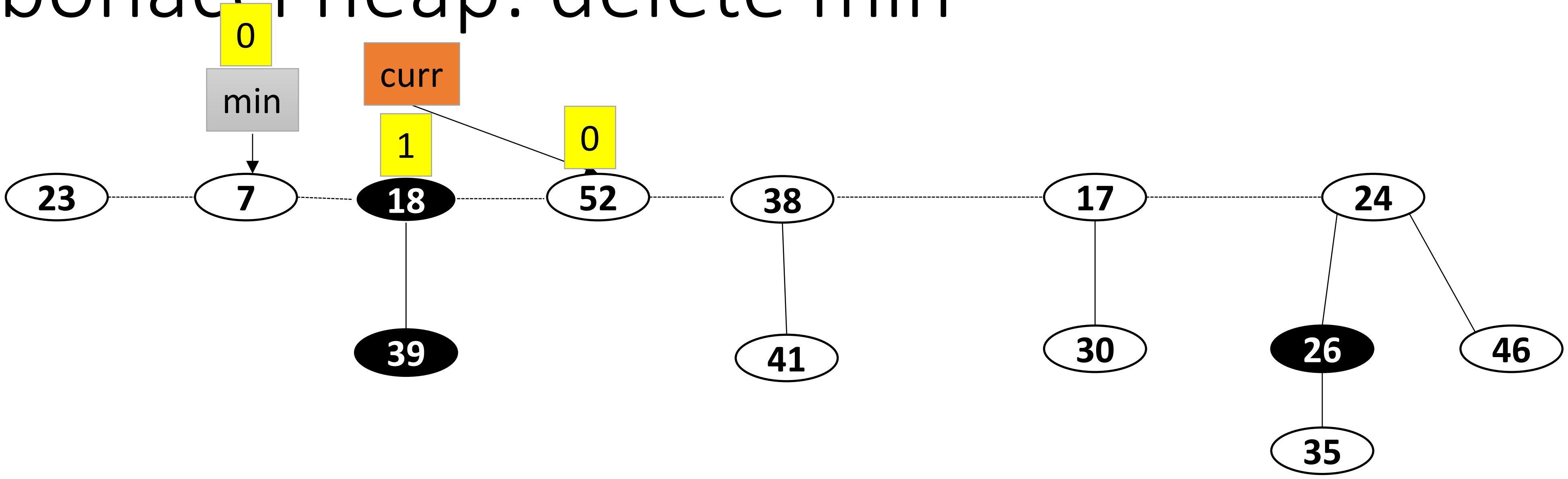




Steps:

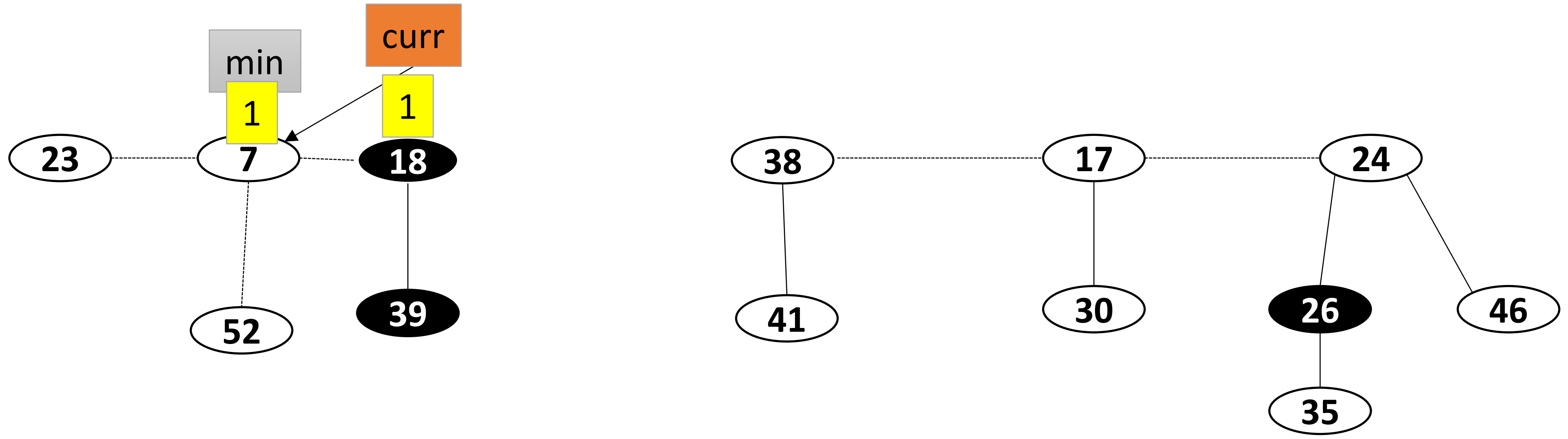
1) Delete min and concatenates the children to the root list.
2) Consolidate the trees so that no two roots have the same degree.

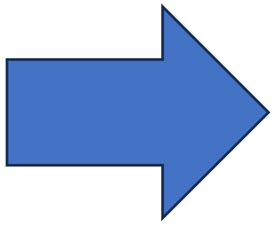
Fibonacci heap: delete min



Steps:
1) Delete min and concatenates the children to the root list.
2) Consolidate the trees so that no two roots have the same degree.

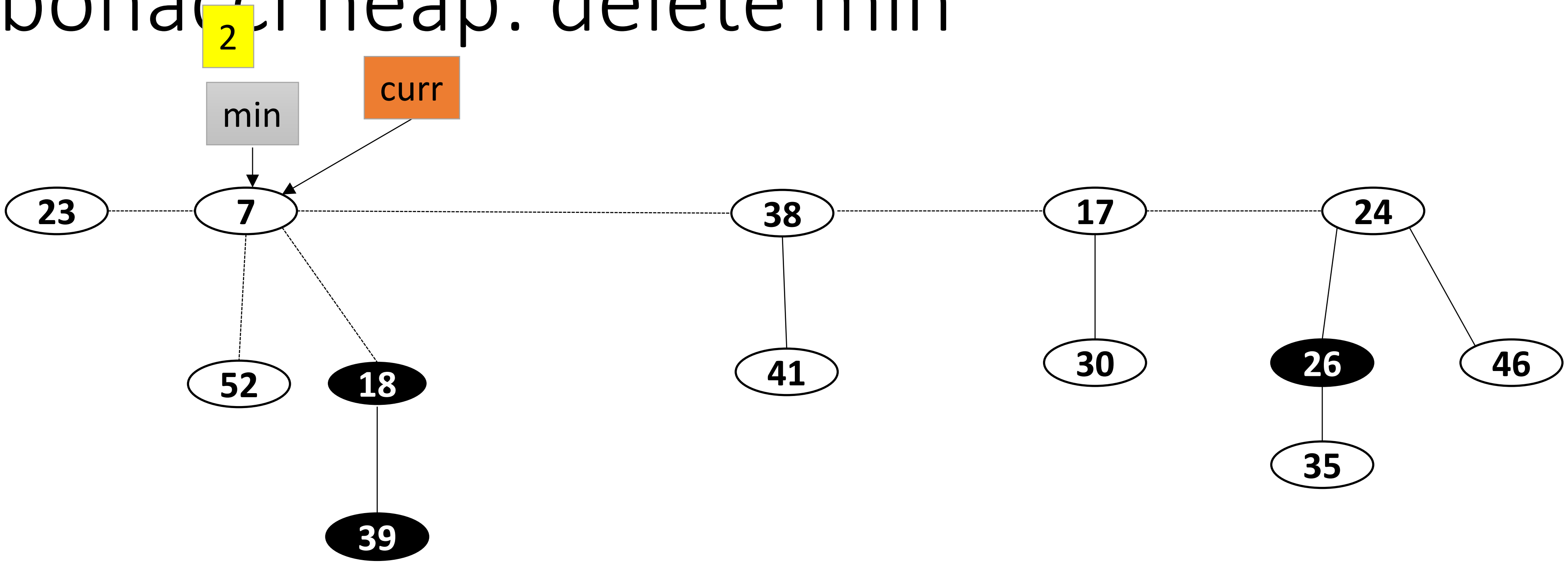
Fibonacci heap: delete min

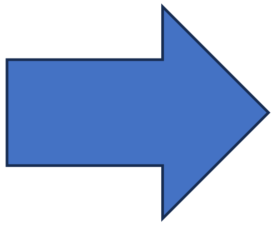




Steps:
1) Delete min and concatenates the children to the root list.
2) Consolidate the trees so that no two roots have the same degree.

Fibonacci heap: delete min

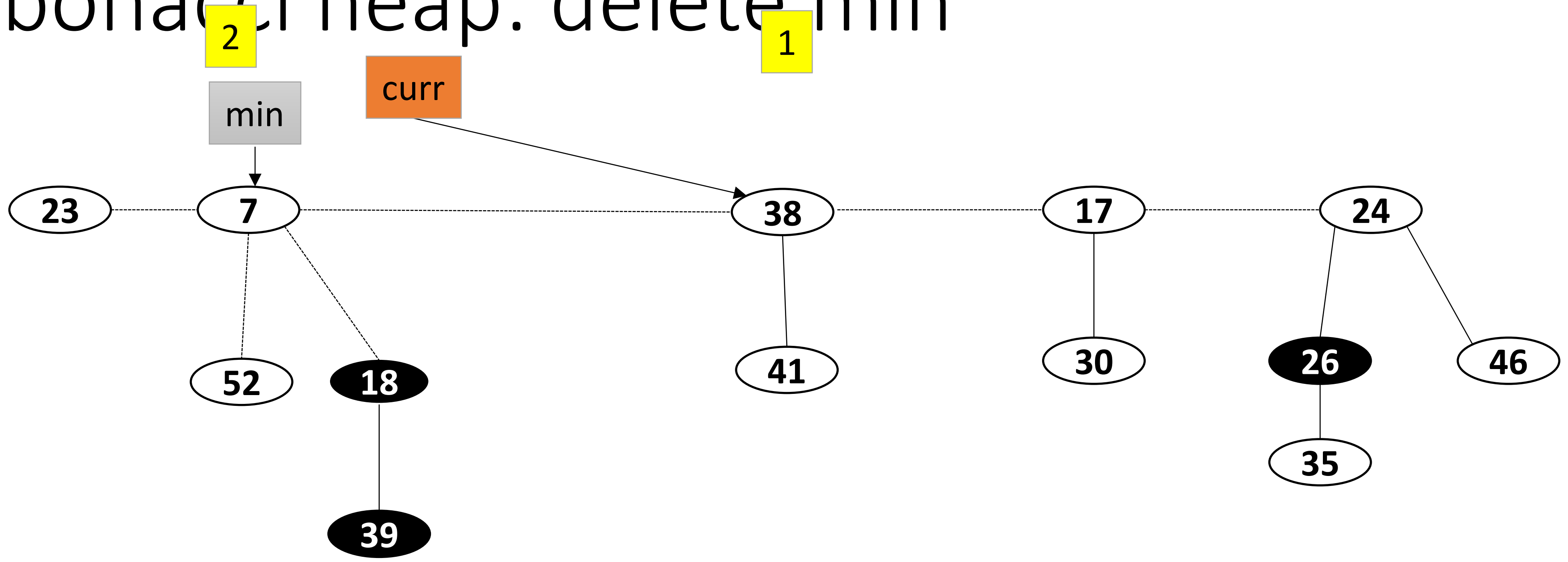




Steps:

1) Delete min and concatenates the children to the root list.
2) Consolidate the trees so that no two roots have the same degree.

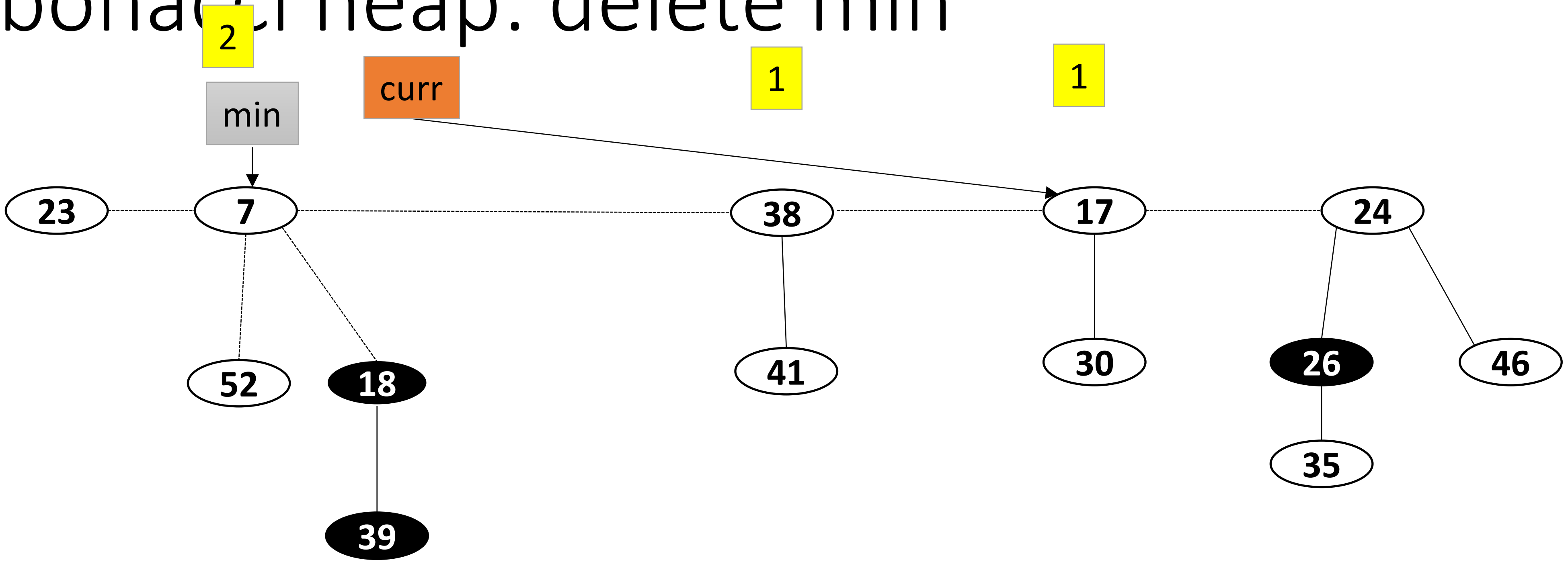
Fibonacci heap: delete min

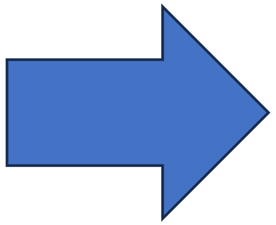


Steps:

1) Delete min and concatenates the children to the root list.
2) Consolidate the trees so that no two roots have the same degree.

Fibonacci heap: delete min

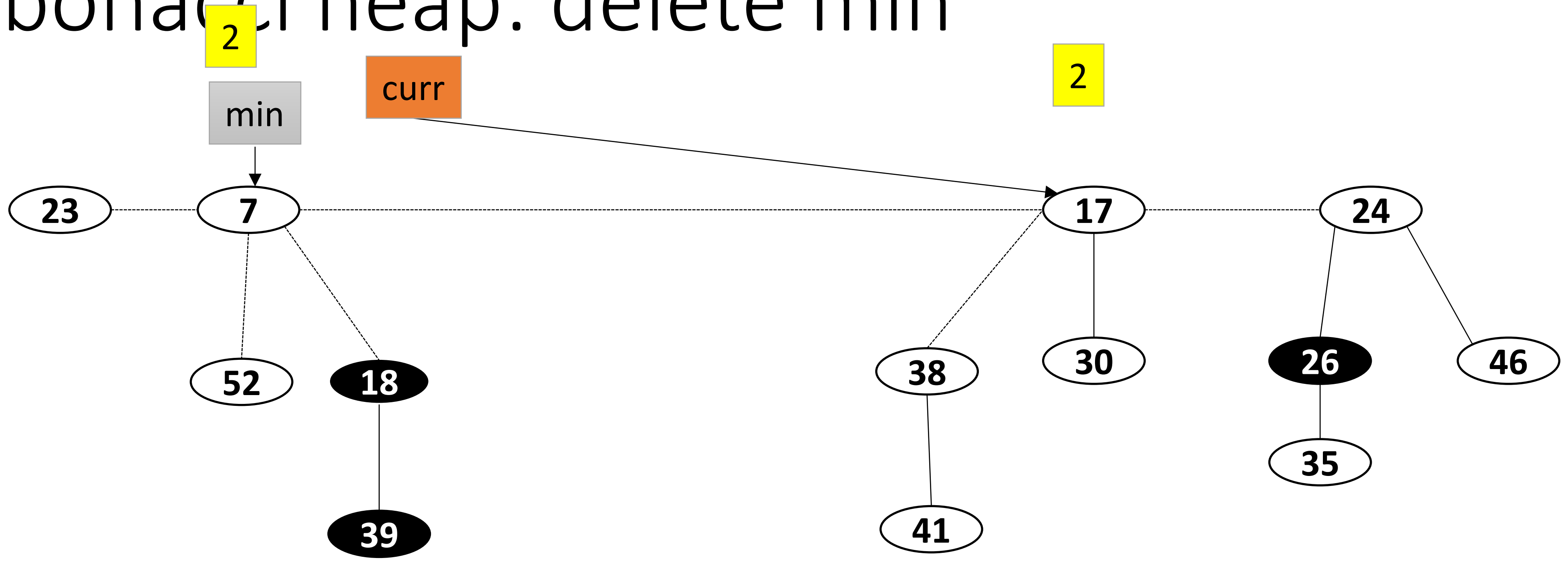




Steps:

1) Delete min and concatenates the children to the root list.
2) Consolidate the trees so that no two roots have the same degree.

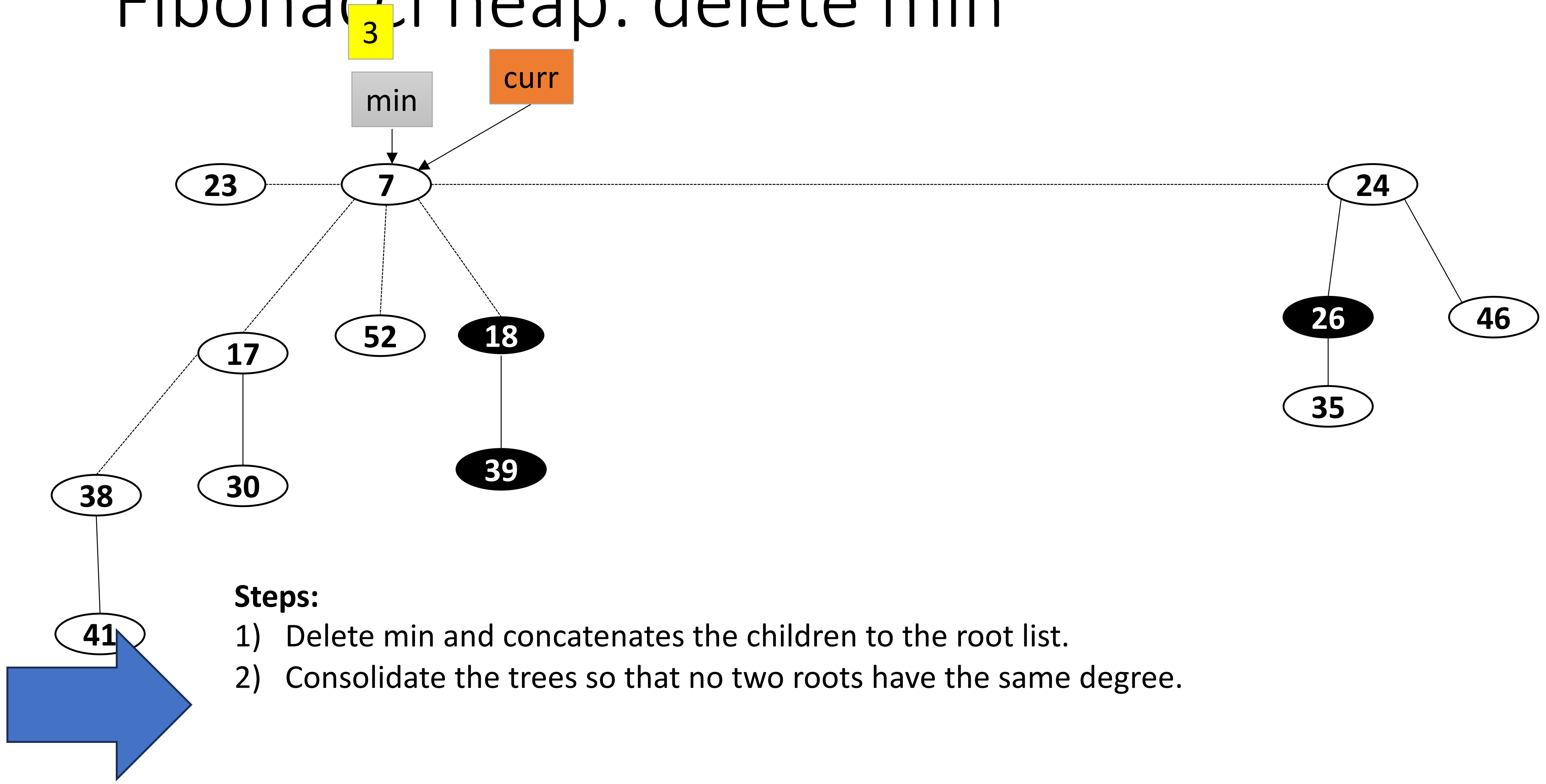
Fibonacci heap: delete min



Steps:

1) Delete min and concatenates the children to the root list.
2) Consolidate the trees so that no two roots have the same degree.

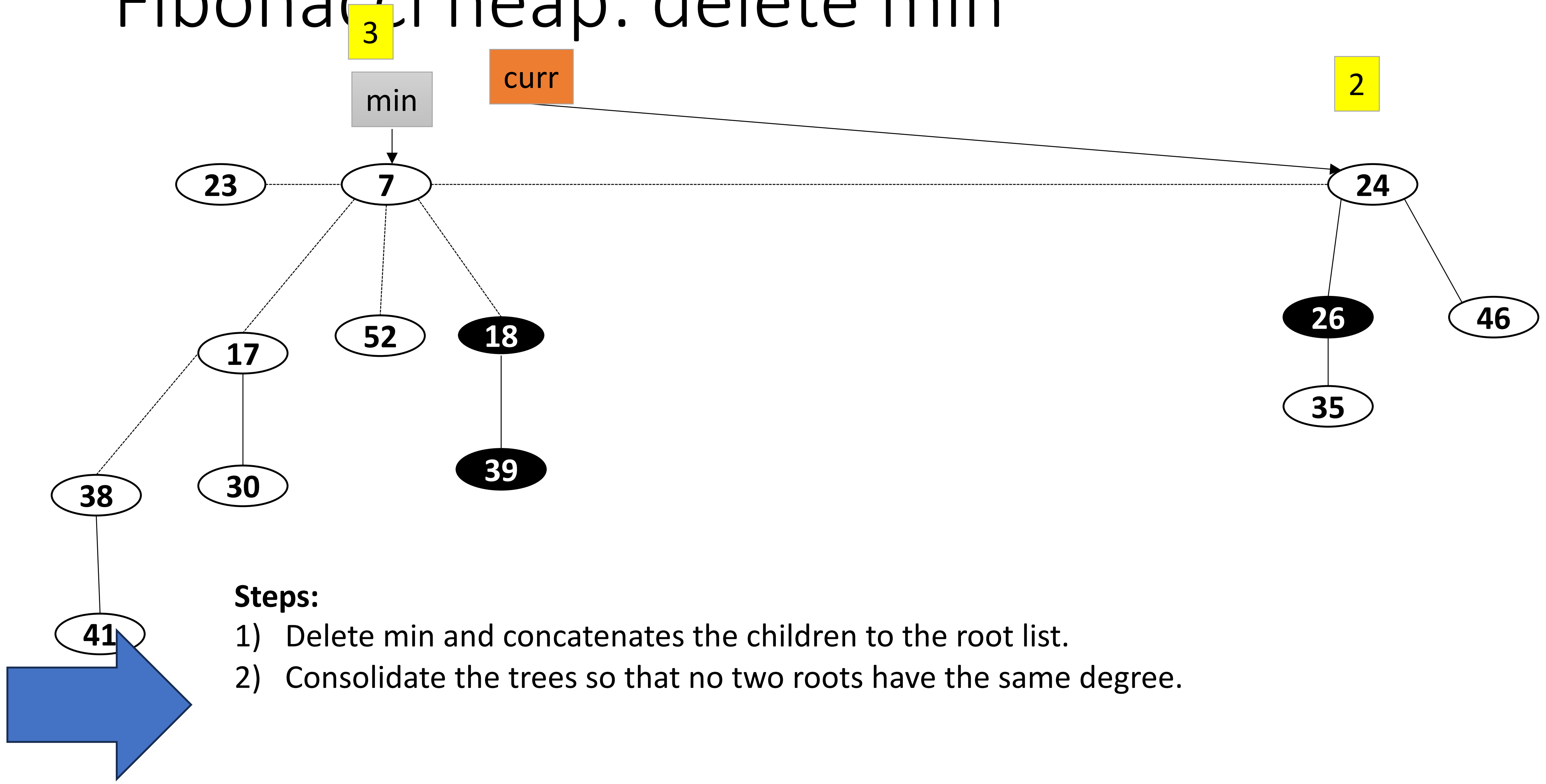
Fibonacci heap: delete min



Steps:

1) Delete min and concatenates the children to the root list.
2) Consolidate the trees so that no two roots have the same degree.

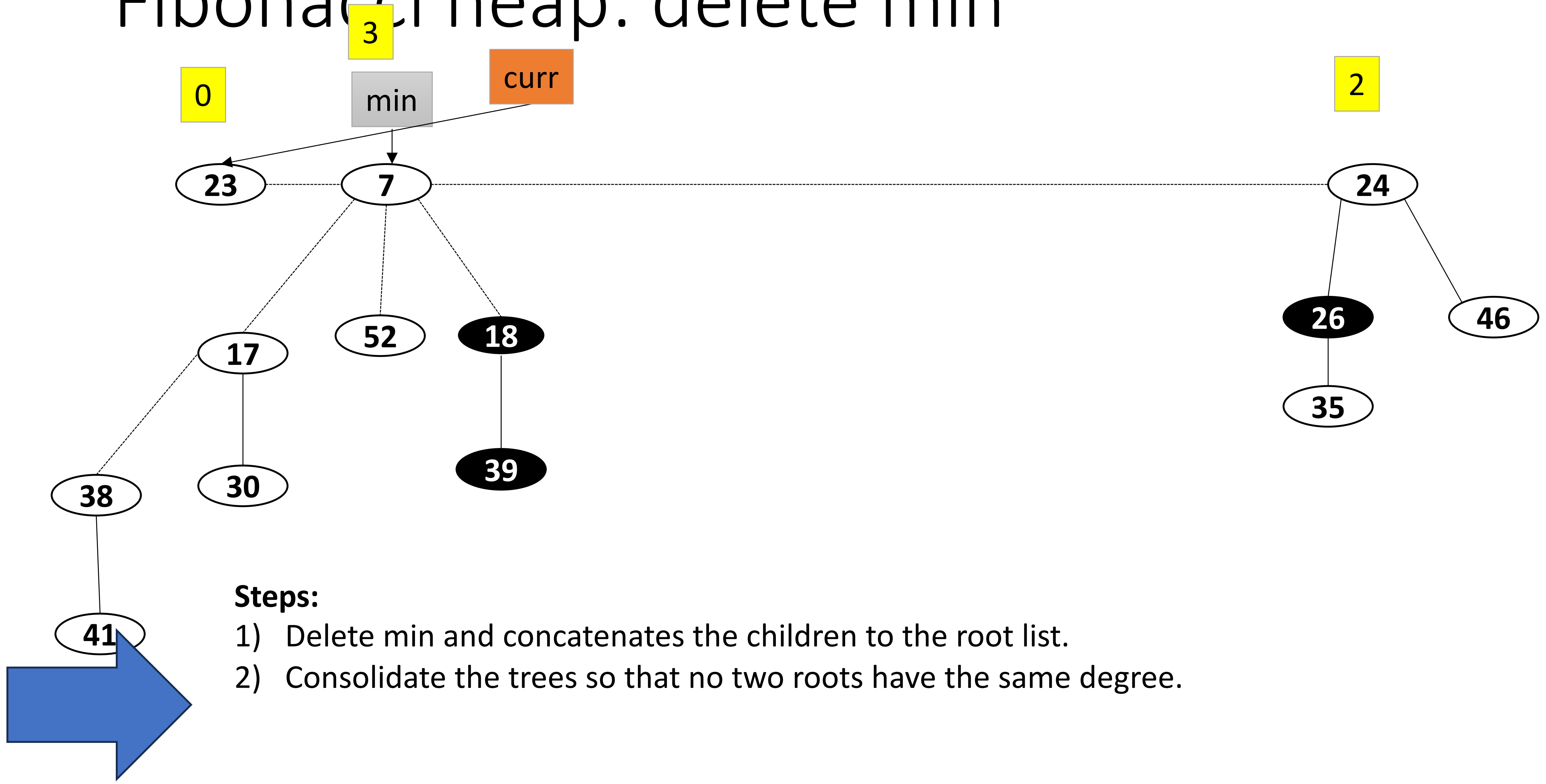
Fibonacci heap: delete min



Steps:

1) Delete min and concatenates the children to the root list.
2) Consolidate the trees so that no two roots have the same degree.

Fibonacci heap: delete min



Steps:

1) Delete min and concatenates the children to the root list.
2) Consolidate the trees so that no two roots have the same degree.

Complexity Analysis

Operation	Binary Heap (worst case)	Fibonacci Heap (amortized)
Make-Heap	O(1)	O(1)
Insert	O(log n)	O(1)
Minimum	O(1)	O(1)
Extract-Min	O(log n)	O(log n)
Union	O(n)	O(1)
Decrease-Key	O(log n)	O(1)
Delete	O(log n)	O(log n)

Note: Fibonacci Heaps are desirable when the number of ***Extract-Min*** and ***Delete*** operations is small relative to the number of other operations.

Mergeable heaps

- Fibonacci Heaps also supports the following operations:
  - Decrease-Key(H,x,k): assigns to element x within heap H the new key value k, which is assumed to be no greater than its current key value.
  - Delete(H,x): delete element x from the heap H.
- ***Note***: It is also possible to implement mergeable max-heap with *Maximum, Extract-Max*, and *Increase-Key* operations.

Applications of Fibonacci Heaps

- Implementing fast algorithms for:
  - Computing minimum spanning trees
  - Finding single-source shortest paths

- Have more theoretical than practical appeal due to programming complexity.

Source

- Introduction to Algorithms- 3rd Edition, Cormen et al (2009)