Amortized analysis

Petr Kurapov

Fall 2024

Amortized complexity

- Worst-case complexity – too pessimistic
- “Online” algorithms, ie. std::vector
- Pushing $n + 1$ elements to a vector of size n : each push costs $\Theta(1)$, last takes $\Theta(n)$ → average $\frac{\Theta(n)+n\Theta(1)}{n+1} = \Theta(1)$

Amortized complexity

- Total expense per operation over an operation sequence: $\frac{T(n)}{n}$
- Permit rare expensive operations while guaranteeing total cost (asymptotic worst-case)
- Amortized analysis:
  - Aggregate analysis (upper bound for $T(n)$ for a sequence of n operations, analyze average worst-case time)
  - Accounting method (assign credit for ops for latter use, may differ from operations' actual cost)
  - Potential method (immediate cost + potential change)

Tarjan. Amortized computational complexity. https://www2.cs.duke.edu/courses/fall11/cps234/reading/Tarjan85_AmortizedComplexity.pdf

Accounting method

- c_i - true cost, c_i' - charge for operation : $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} c_i'$
- $c_i = \begin{cases} i, if\ i - 1 = 2^k \\ 1 \end{cases}$

i	1	2	3	4	5	6	7	8	9	10
Capacity	1	2	4	4	8	8	8	8	16	16
Cost	1	2	3	1	5	1	1	1	9	1
Charge										
Balance										

Accounting method

- c_i - true cost, c_i' - charge for operation : $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} c_i'$
- $c_i = \begin{cases} i, if\ i-1 = 2^k \\ \quad\quad 1 \end{cases}$

i	1	2	3	4	5	6	7	8	9	10
Capacity	1	2	4	4	8	8	8	8	16	16
Cost	1	2	3	1	5	1	1	1	9	1
Charge	3	3	3	3	3	3	3	3	3	3
Balance	3-1=2	3	3	5	3	5	7	9	3	4

1 – immediate insertion, 1 – to move inserted element the first time array is growing, 1 – donated to element $i - 2^k$ to move it

Accounting method

- c_i - true cost, c_i' - charge for operation : $\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} c_i'$
- $c_i = \begin{cases} i, if\ i - 1 = 2^k \\ 1 \end{cases}$

$2^k < 6$

i	1	2	3	4	5	6	7	8	9	10
Capacity	1	2	4	4	8	8	8	8	16	16
Cost	1	2	3	1	5	1	1	1	9	1
Charge	3	3	3	3	3	3	3	3	3	3
Balance	3-1=2	3	3	5	3	5	7	9	3	4

1 – immediate insertion, 1 – to move inserted element the first time array is growing, 1 – donated to element $i - 2^k$ to move it

Potential method

- Potential function Φ: $\Phi(h_0) = 0, \forall h_k$: $\Phi(h_k) \geq 0$. h_k - data structure state
- Analogues to the *balance* in accounting method but is a function of the current data structure state.
- Amortized operation time is actual cost + potential change:
  - $c' = c + \Phi(h') - \Phi(h)$

$$\big(c_0 + \Phi(h_1) - \Phi(h_0)\big) + \big(c_1 + \Phi(h_2) - \Phi(h_1)\big)$$
$$= c_0 + c_1 + \Phi(h_2) - \Phi(h_0) = c_0 + c_1 + \Phi(h_2)$$

Potential method

- For vector: $\Phi(h) = 2n - m$, n – number of elements, m – capacity.
- 2 cases:
- $n < m, cost = 1$ (n++); potential change $(2(n+1) - m) - (2n - m) = 2 \to$ amortized time $1 + 2 = 3$
- $n = m$, $cost = n + 1$; potential change $(2(n+1) - 2n) - (2n - n) = 2 - n \to$ amortized time $n + 1 + (2 - n) = 3$

Example: Priority queue

- Set of elements, each associated with a key. Supports:
- Inserting new elements
- Get element with max key
- Pop max
- Increase element’s key
- Merge*
- Delete*

Priority queue

- Why use?

Priority queue

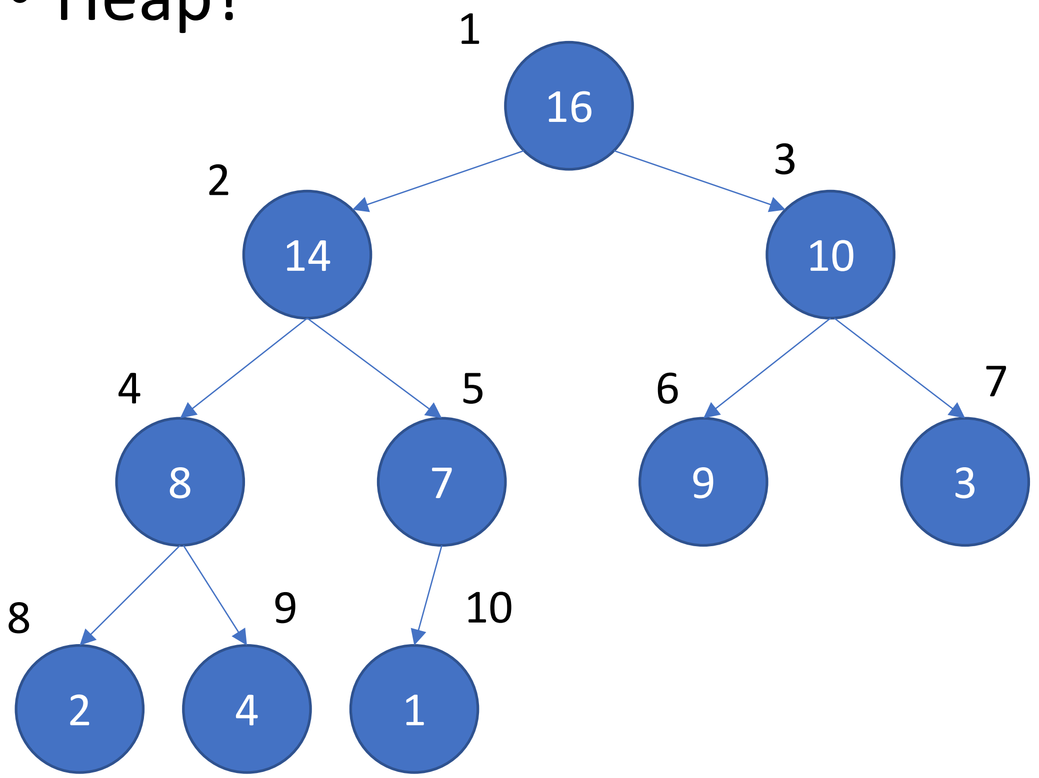
- Why use?
- Resource allocation scheduling
- Minimum spanning tree (Prim’s algorithm)
- Real-time Optimally Adapting Meshes (ROAM) – triangulation
- Dijkstra and A* algorithms
- …

Priority queue: implementation

- Insert
- Get max
- Pop max
- Increase key
- Merge*
- Delete*

Priority queue: implementation

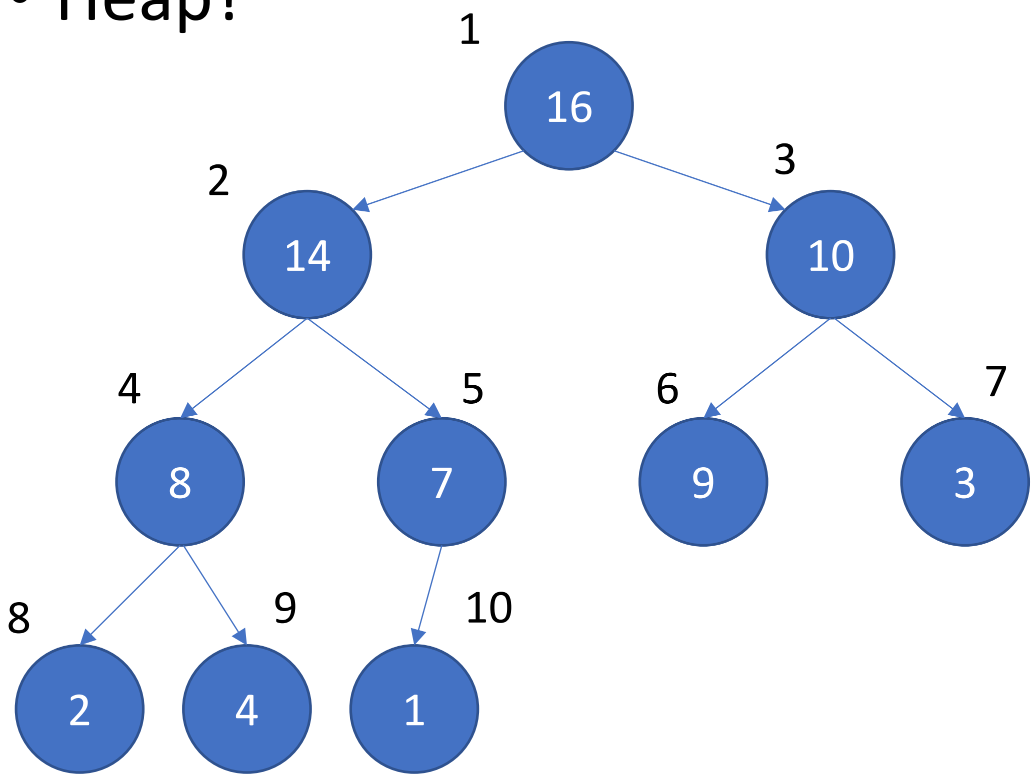
- Heap?



- Insert
- Get max
- Pop max
- Increase key
- Merge*
- Delete*

Priority queue: implementation

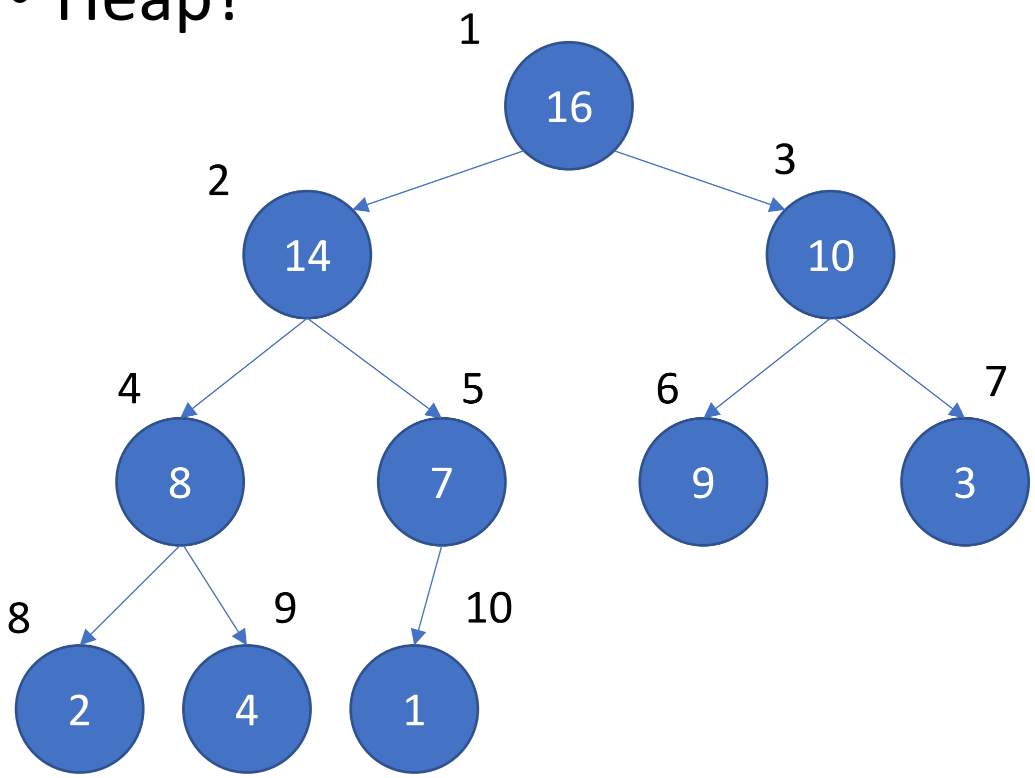
- Heap?



Method	complexity
Insert	
Get max	
Pop max	
Increase key	
Merge*	

Priority queue: implementation

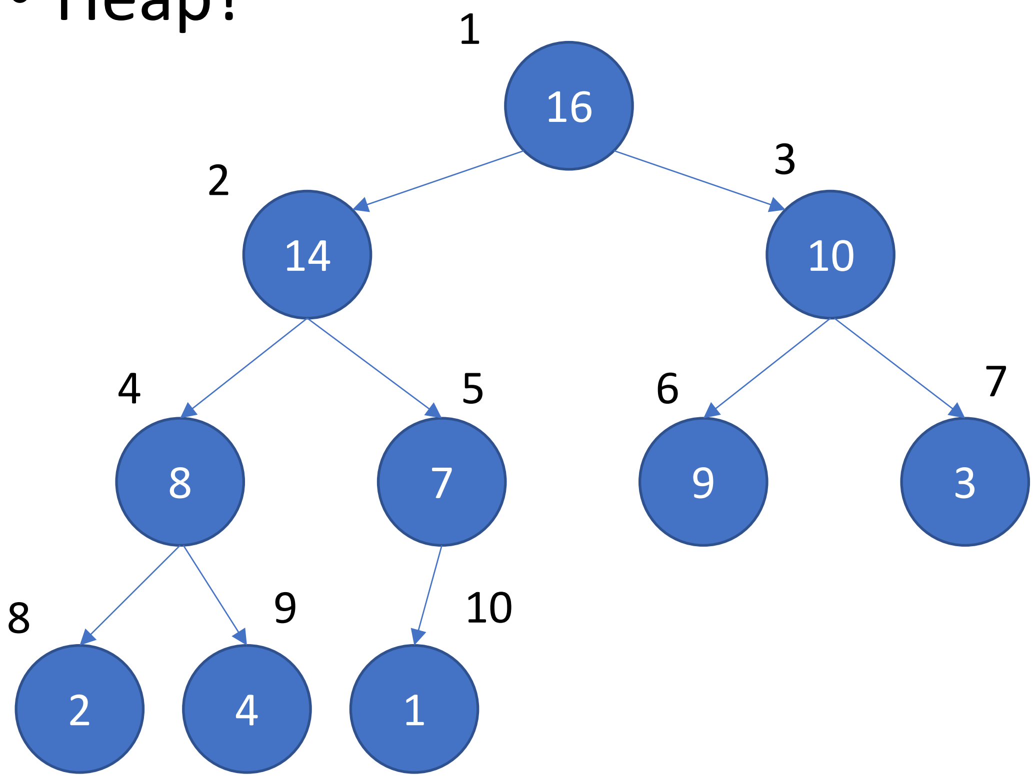
- Heap?



Method	complexity
Insert	$\Theta(\log n)$
Get max	
Pop max	
Increase key	
Merge*	

Priority queue: implementation

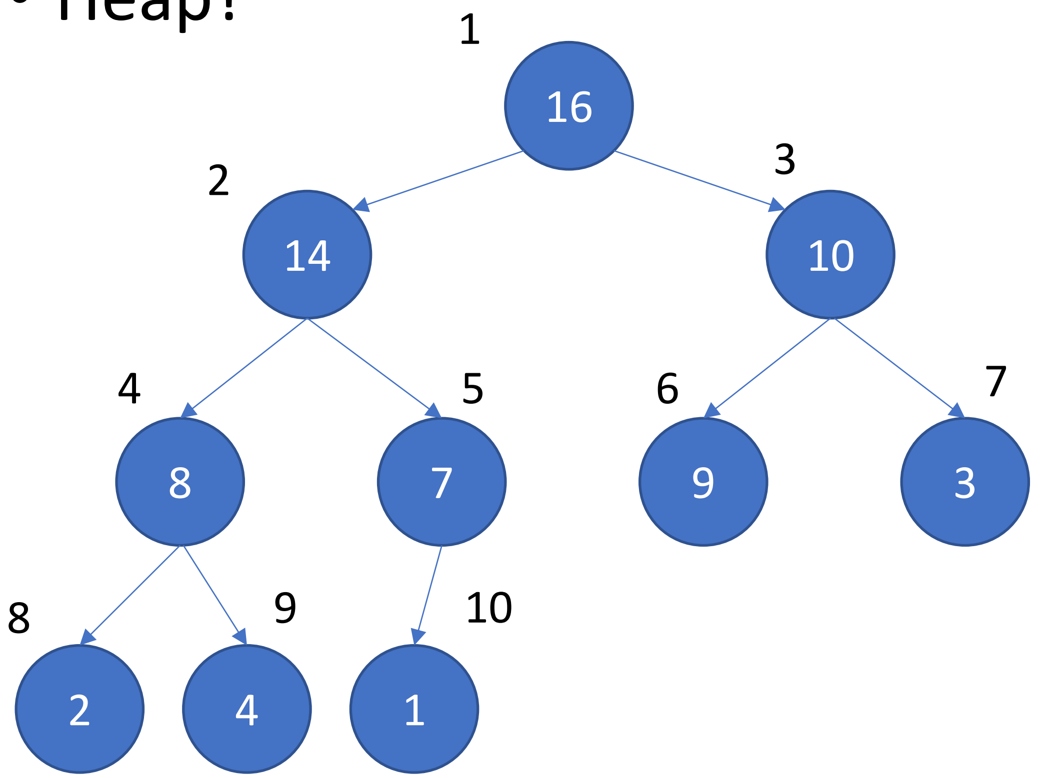
- Heap?



Method	complexity
Insert	$\Theta(\log n)$
Get max	$\Theta(1)$
Pop max	
Increase key	
Merge*	

Priority queue: implementation

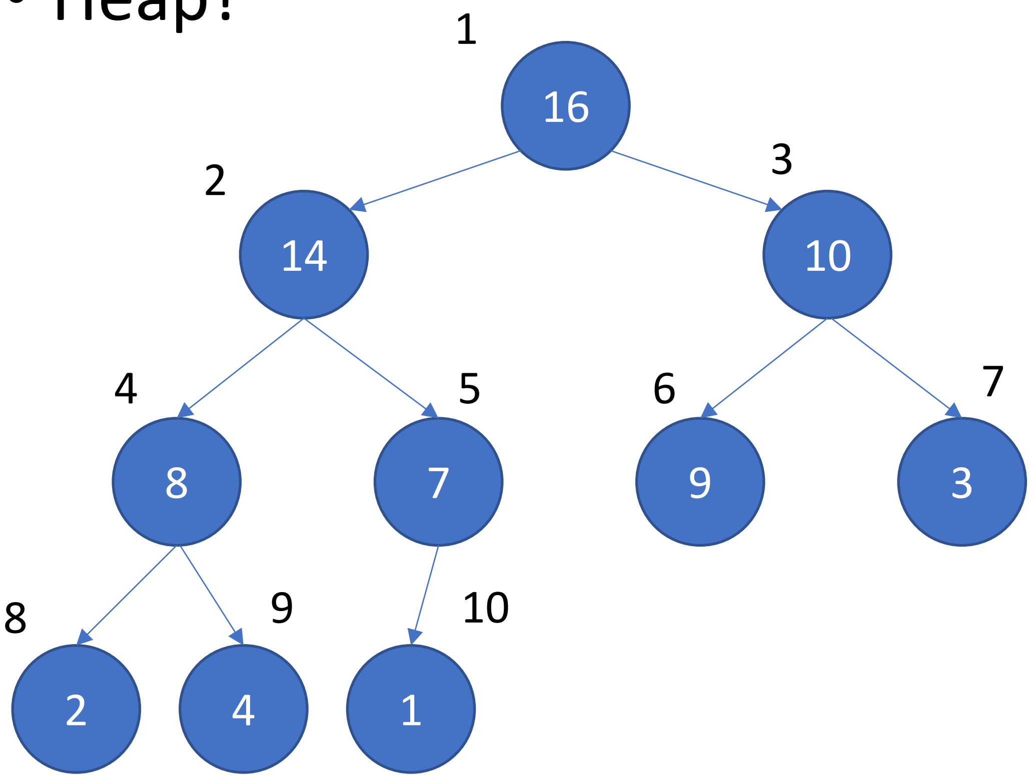
- Heap?



Method	complexity
Insert	$\Theta(\log n)$
Get max	$\Theta(1)$
Pop max	$\Theta(\log n)$
Increase key	
Merge*	

Priority queue: implementation

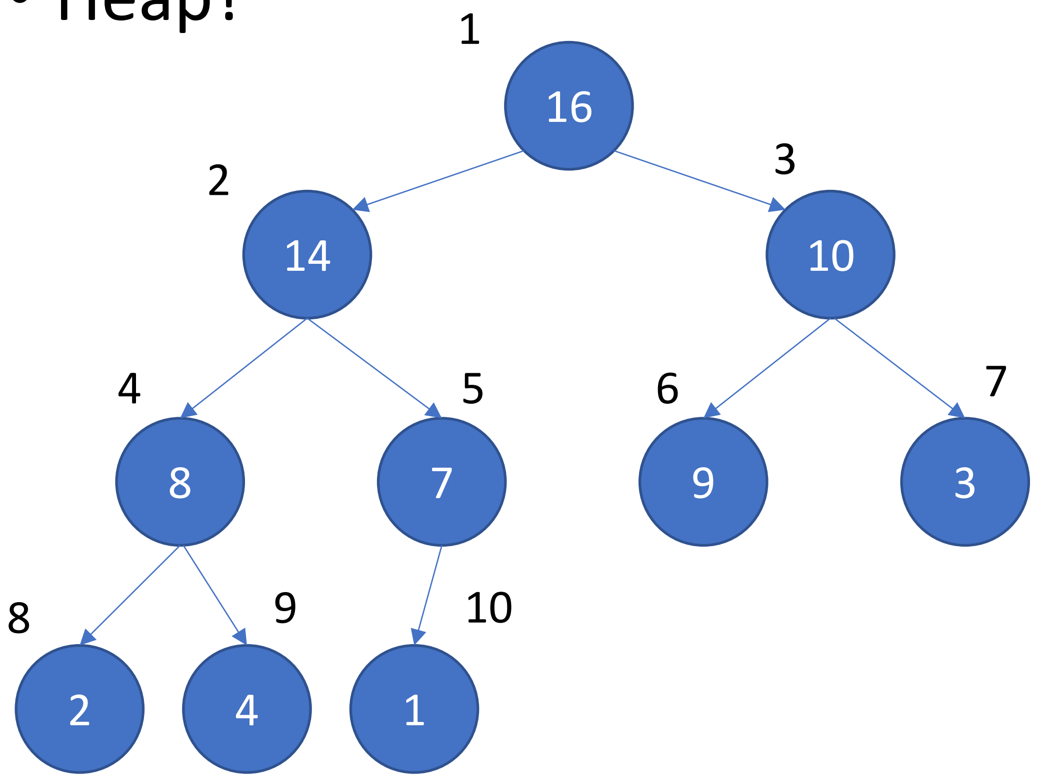
- Heap?



Method	complexity
Insert	$\Theta(\log n)$
Get max	$\Theta(1)$
Pop max	$\Theta(\log n)$
Increase key	$\Theta(\log n)$
Merge*	

Priority queue: implementation

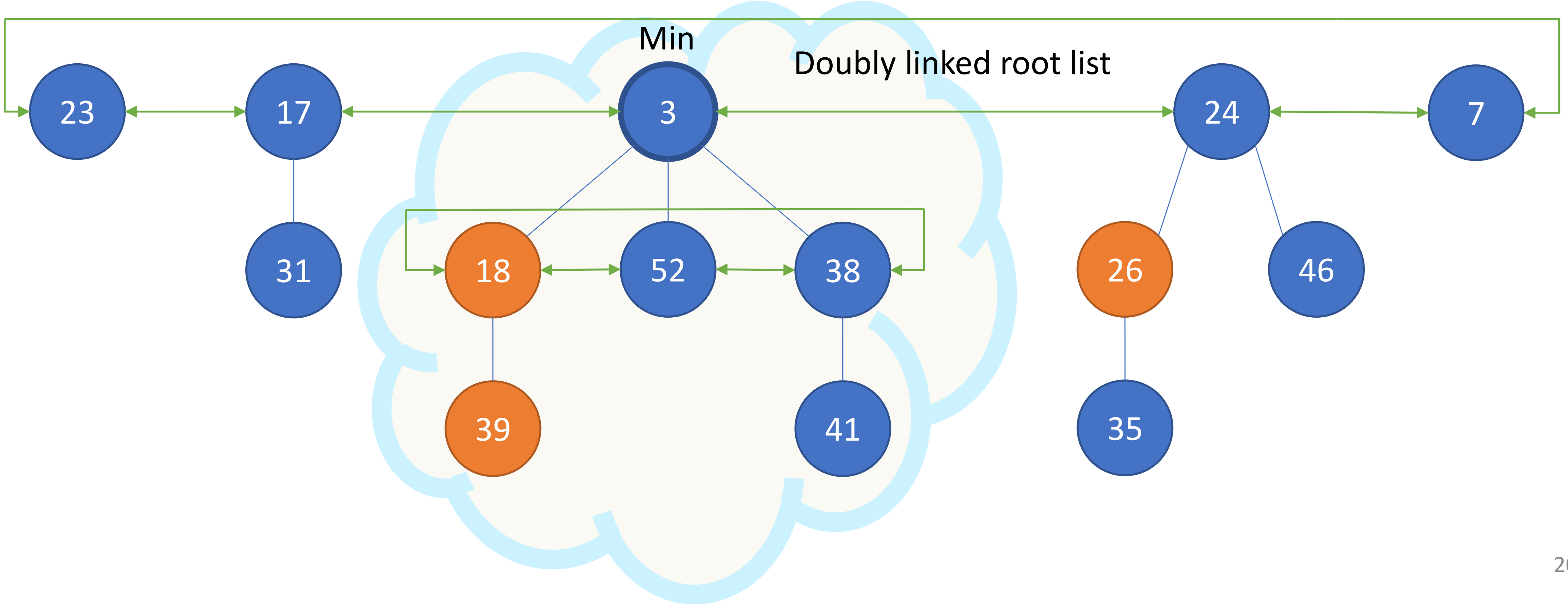
- Heap?



Method	complexity
Insert	$\Theta(\log n)$
Get max	$\Theta(1)$
Pop max	$\Theta(\log n)$
Increase key	$\Theta(\log n)$
Merge*	$\Theta(n)$

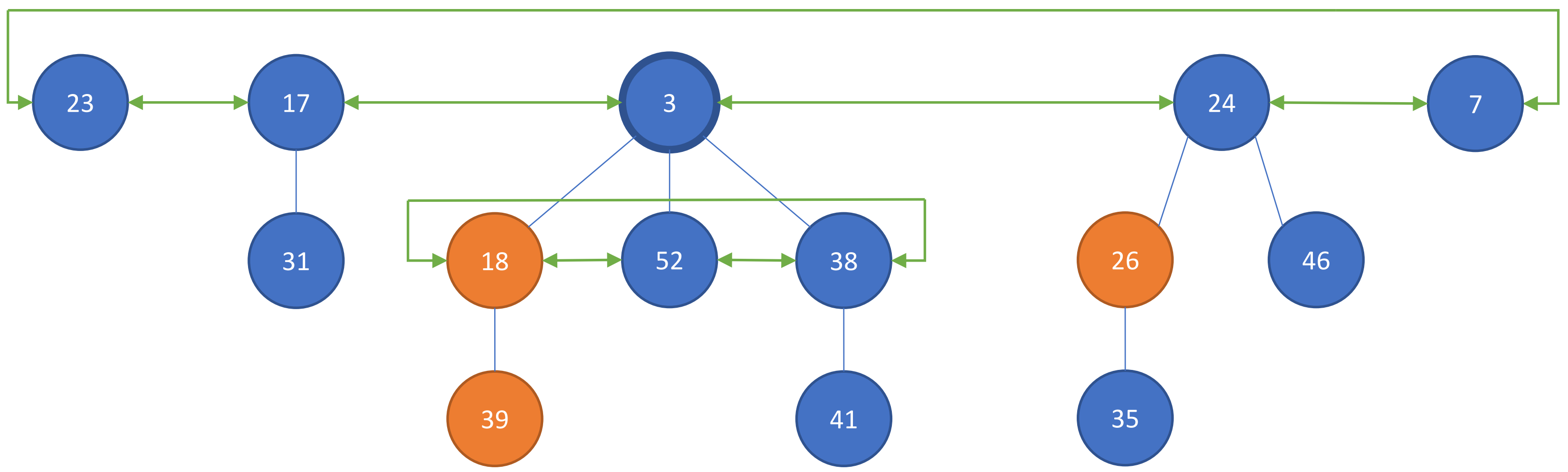
Fibonacci heap

Min-heaps, each node has parent & child pointers + doubly linked list of siblings



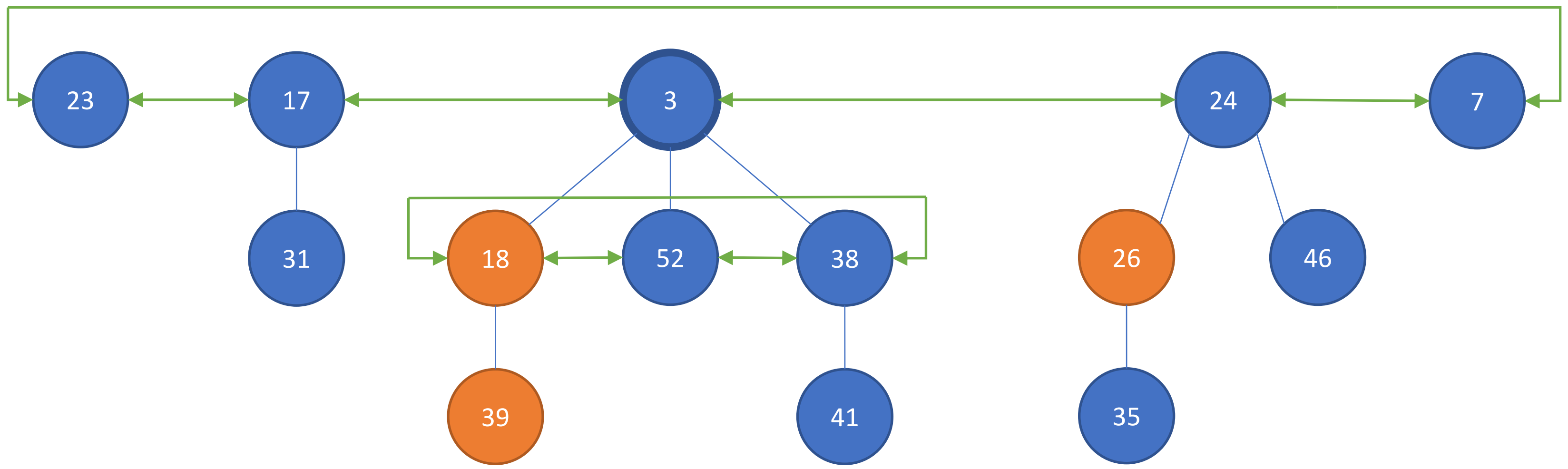
Fibonacci heap

```
struct Node {
    unsigned degree;
    Node* child;
    Node* p;
    bool mark;
    // right & left
};
```



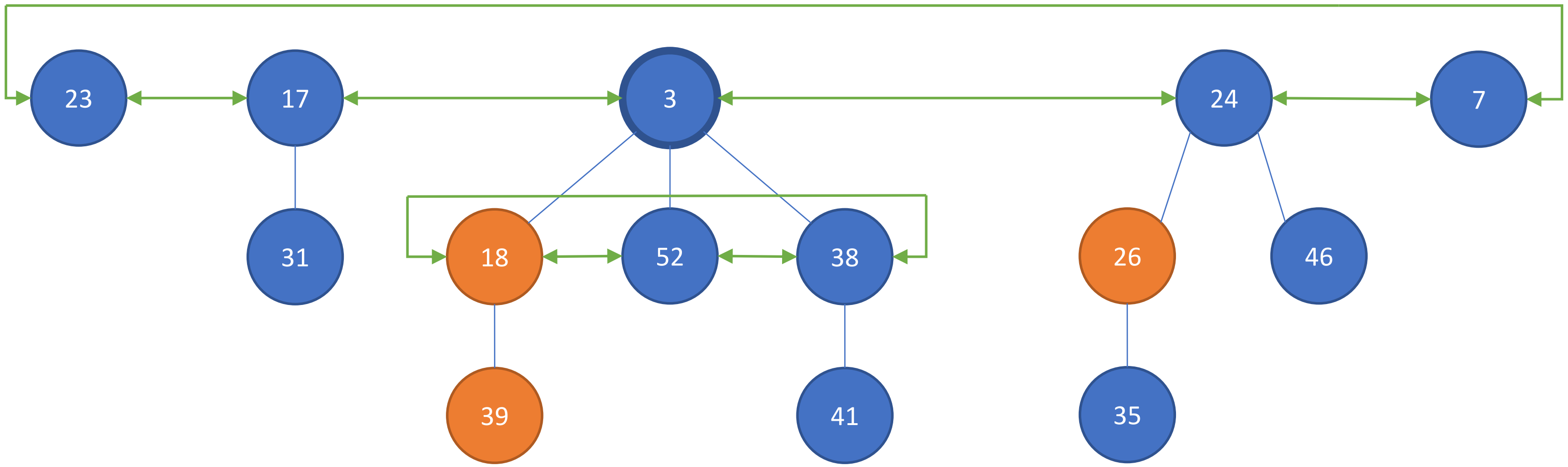
Fibonacci heap

- Insert
- Get min
- Pop min
- Increase key
- Merge



Fibonacci heap: insert

```
void insert(List H, T x)
{
    n = Node(x);
    if (Min == nullptr)
        Min = n;
    else {
        H.insert(n);
        updateMin(n);
    }
}
```

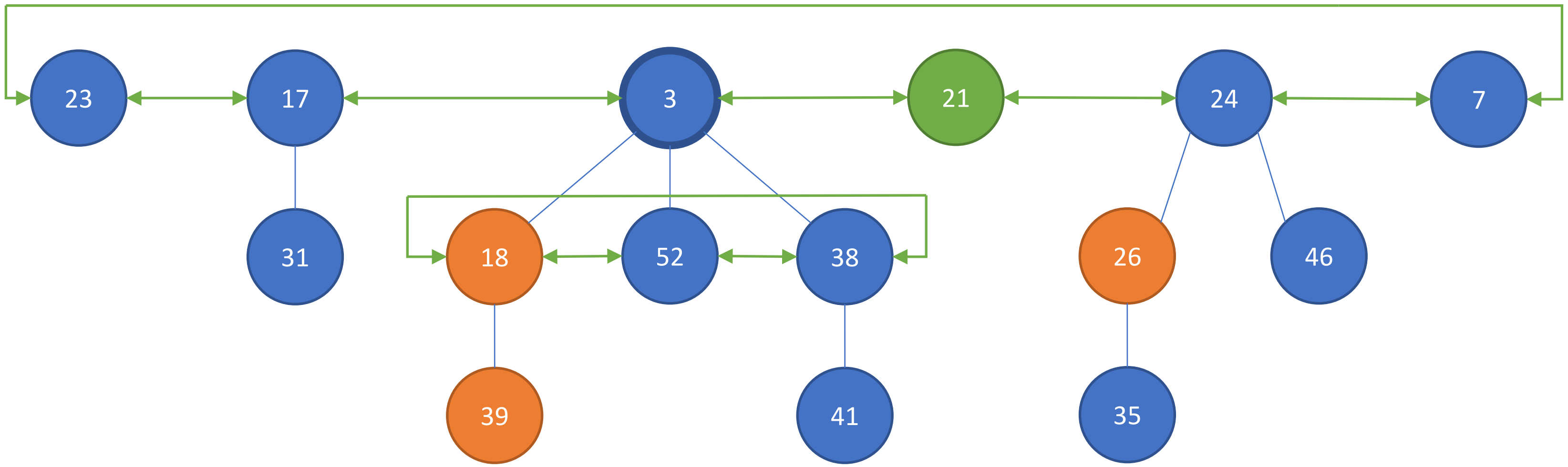


- Lazy consolidation later

Method	complexity
Insert	$\Theta(?)$

Fibonacci heap: insert

```
void insert(List H, T x)
{
    n = Node(x);
    if (Min == nullptr)
        Min = n;
    else {
        H.insert(n);
        updateMin(n);
    }
}
```

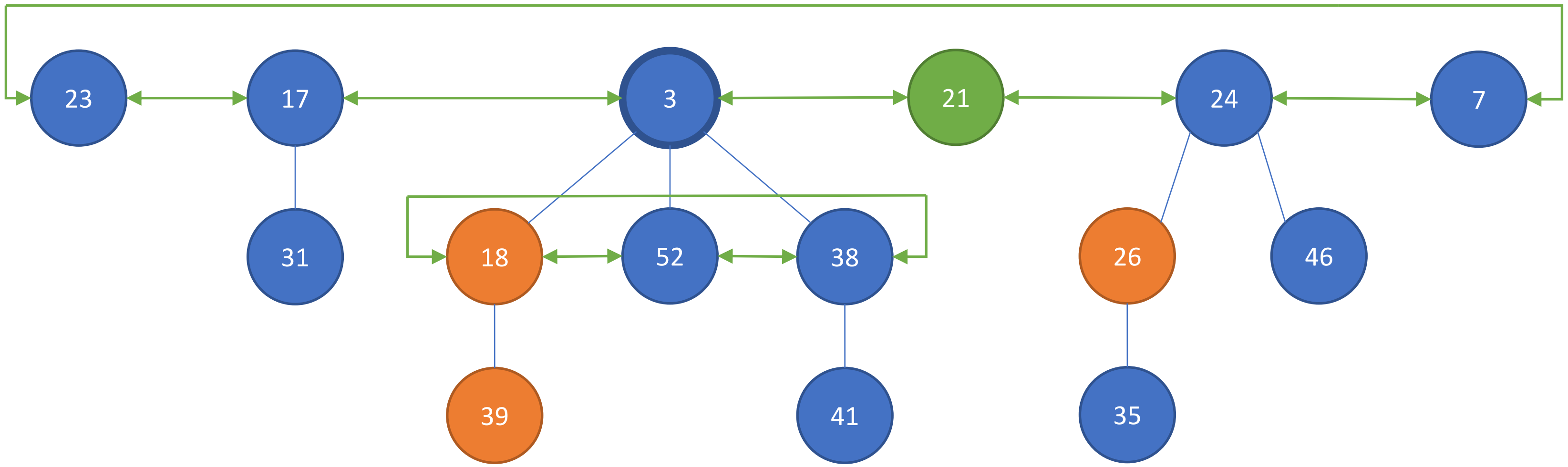


- Lazy consolidation later

Method	complexity
Insert	$\Theta(1)$

Fibonacci heap: insert

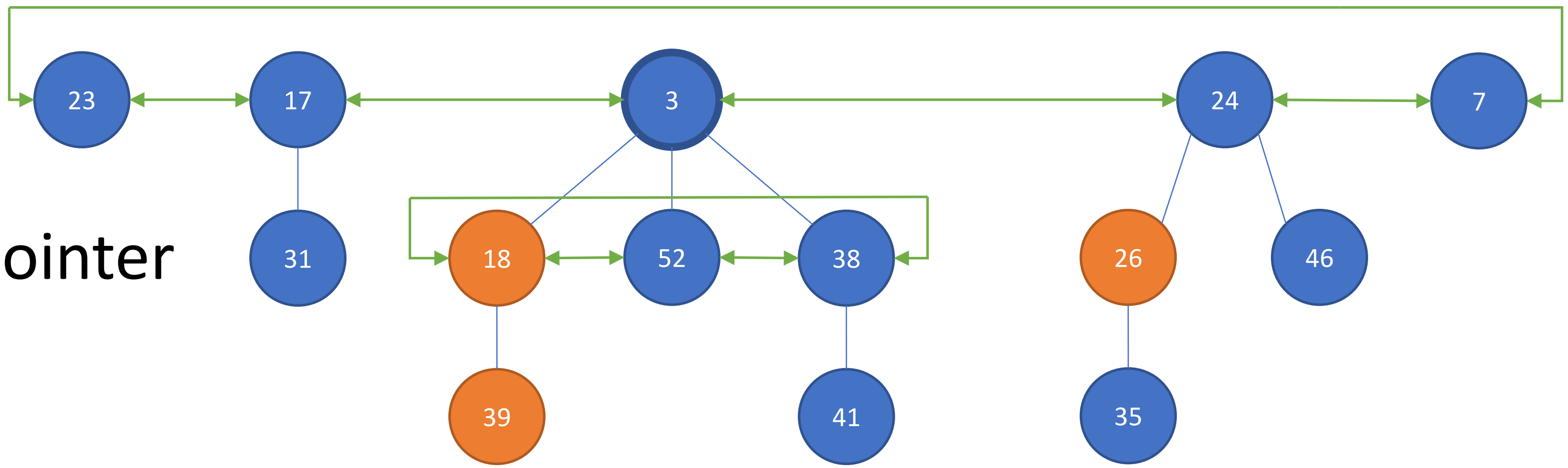
```
void insert(List H, T x)
{
    n = Node(x);
    if (Min == nullptr)
        Min = n;
    else {
        H.insert(n);
        updateMin(n);
    }
}
```



- Lazy consolidation later
- Same for melding – just unify the root list

Fibonacci heap: get min

- Trivial
- The structure stores pointer

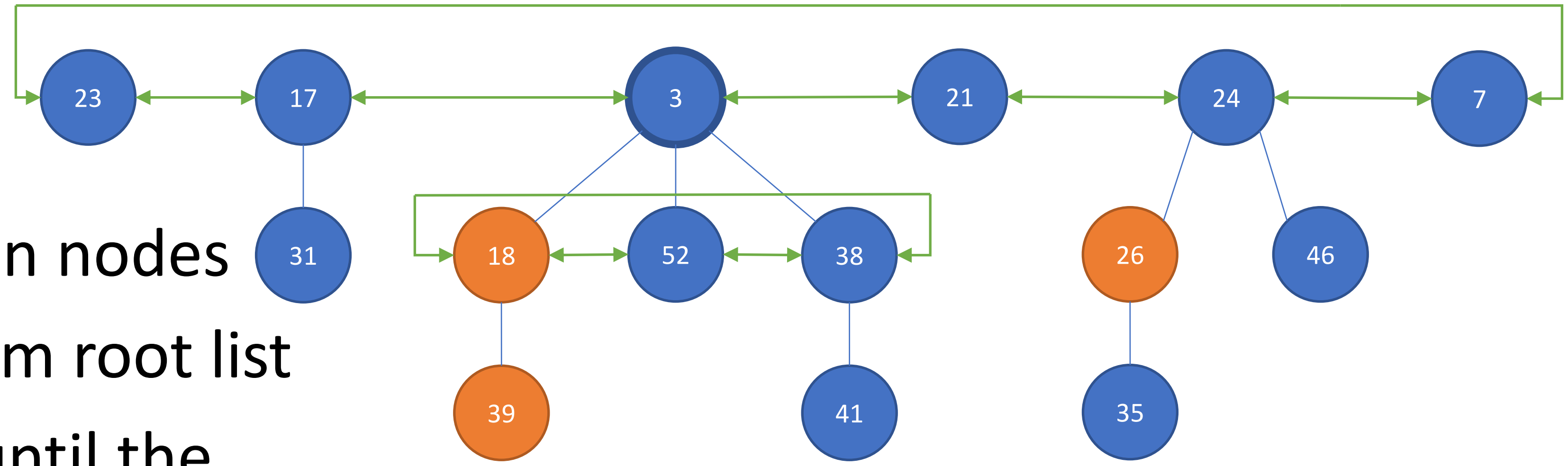


Method	complexity
Get min	$\Theta(1)$

Fibonacci heap: pop min

Idea:

- Create a list of children nodes
- Remove min node from root list
- Consolidate root list until the heap is dense
  - Meld roots of same degree
  - Stop when roots have different degrees



Fibonacci heap: pop min

Idea:

- Create a list of children nodes
- Remove min node from root list
- Consolidate root list until the heap is dense
  - Meld roots of same degree
  - Stop when roots have different degrees

```
Node* pop(List H)
    z = Min; // check if null
    for (x : children(z)) {
        H.append(x);
        x.p = nullptr;
    }
    H.remove(z); // check if empty
    Min = z.right;
    consolidate(H);
    return z;
```

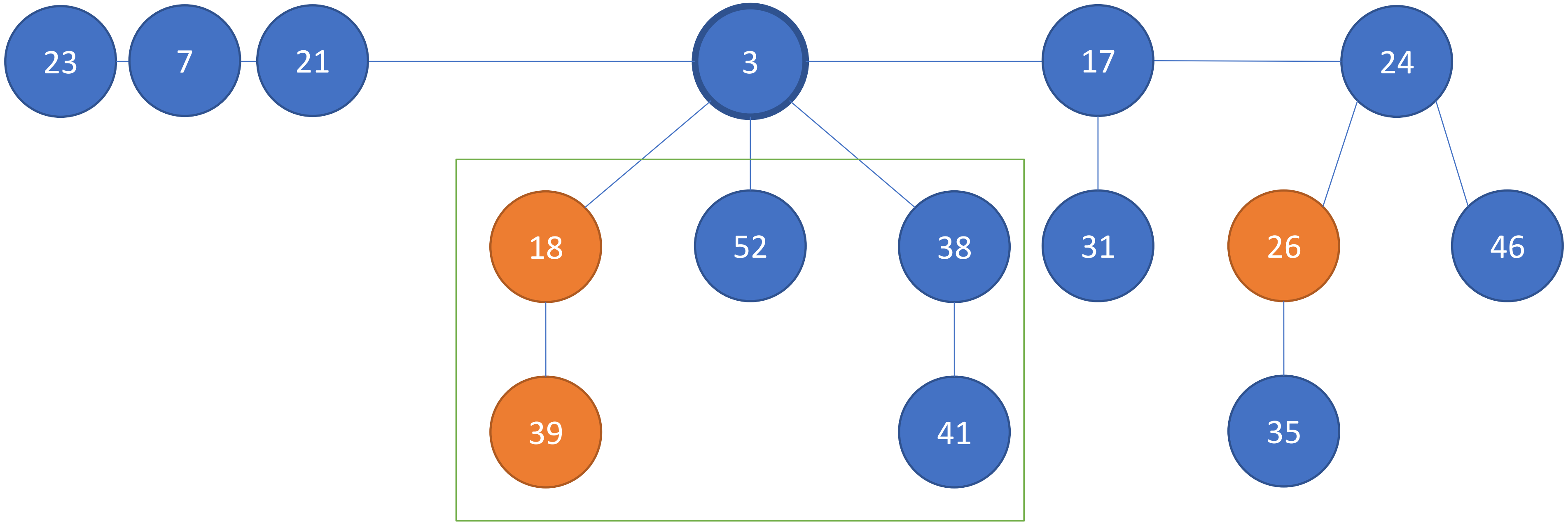
Fibonacci heap: pop min

Consolidation:

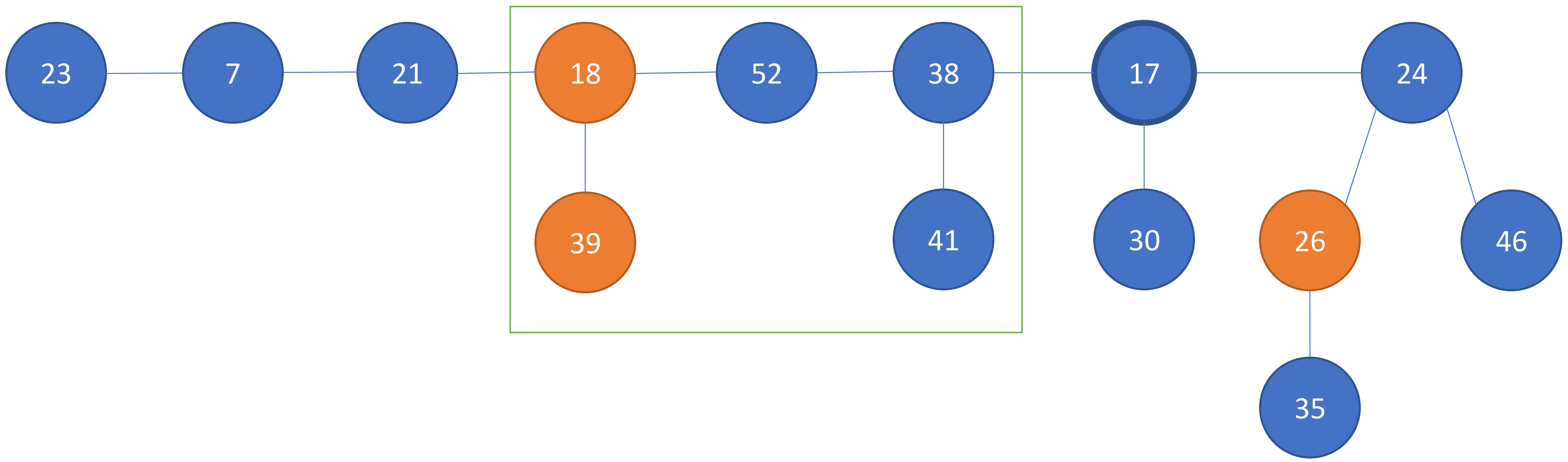
- Find 2 roots of same degree (x.key < y.key)
- Link x&y: remove y from root list, add it to x's children
  - X.degree++
  - Unmark Y

```
Node* pop(List H)
    z = Min; // check if null
    for (x : children(z)) {
        H.append(x);
        x.p = nullptr;
    }
    H.remove(z); // check if empty
    Min = z.right;
    consolidate(H);
    return z;
```

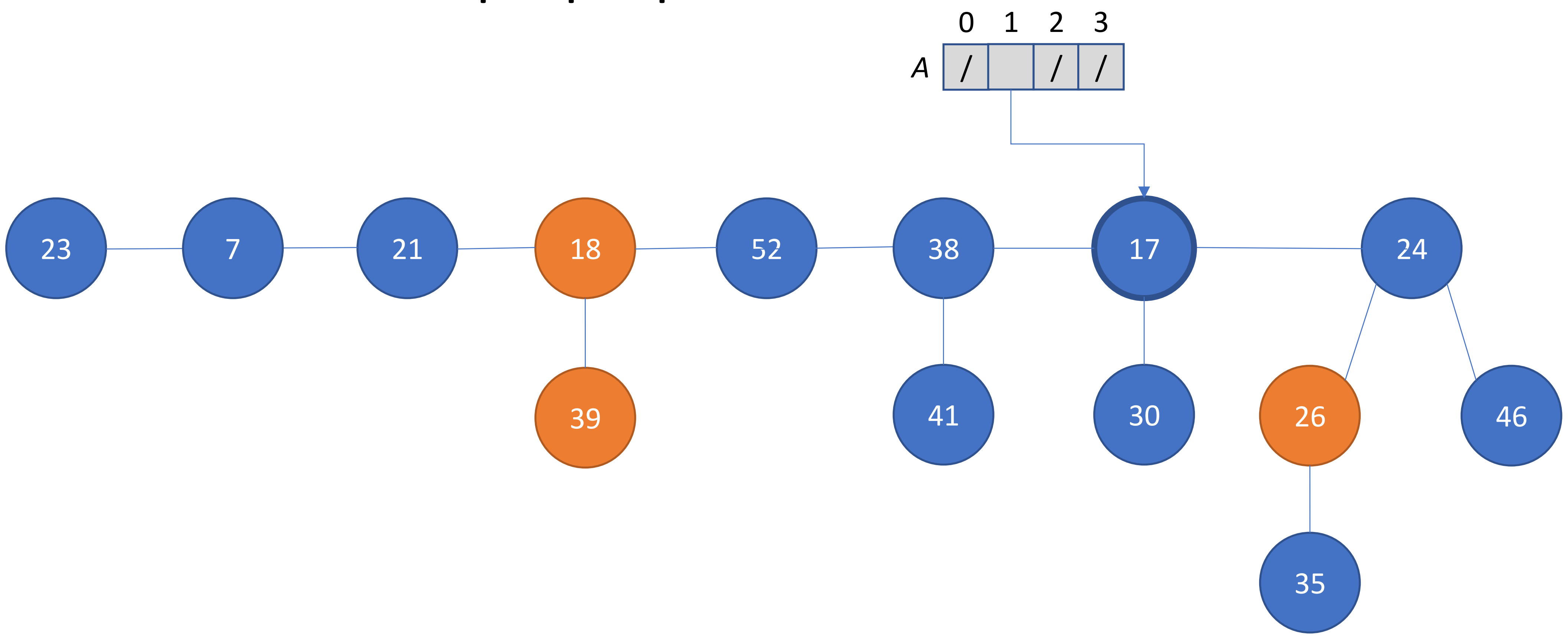
Fibonacci heap: pop min



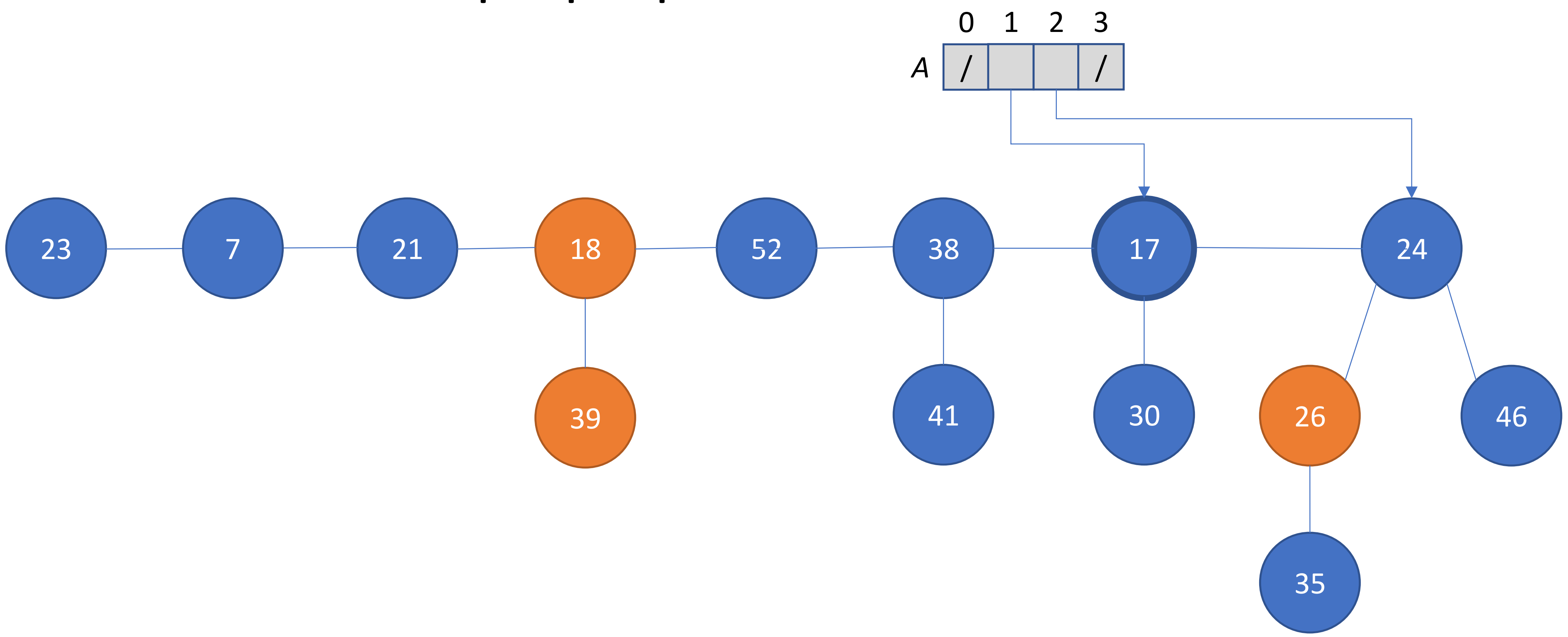
Fibonacci heap: pop min



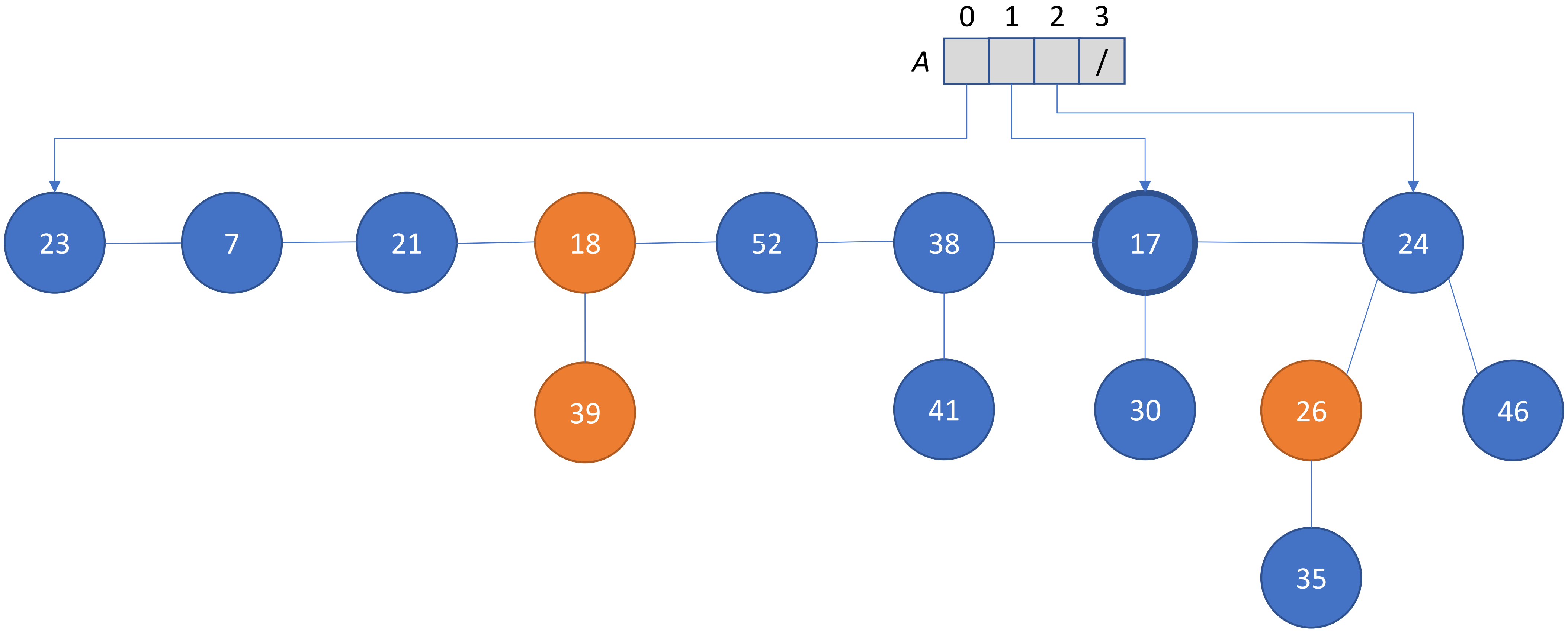
Fibonacci heap: pop min



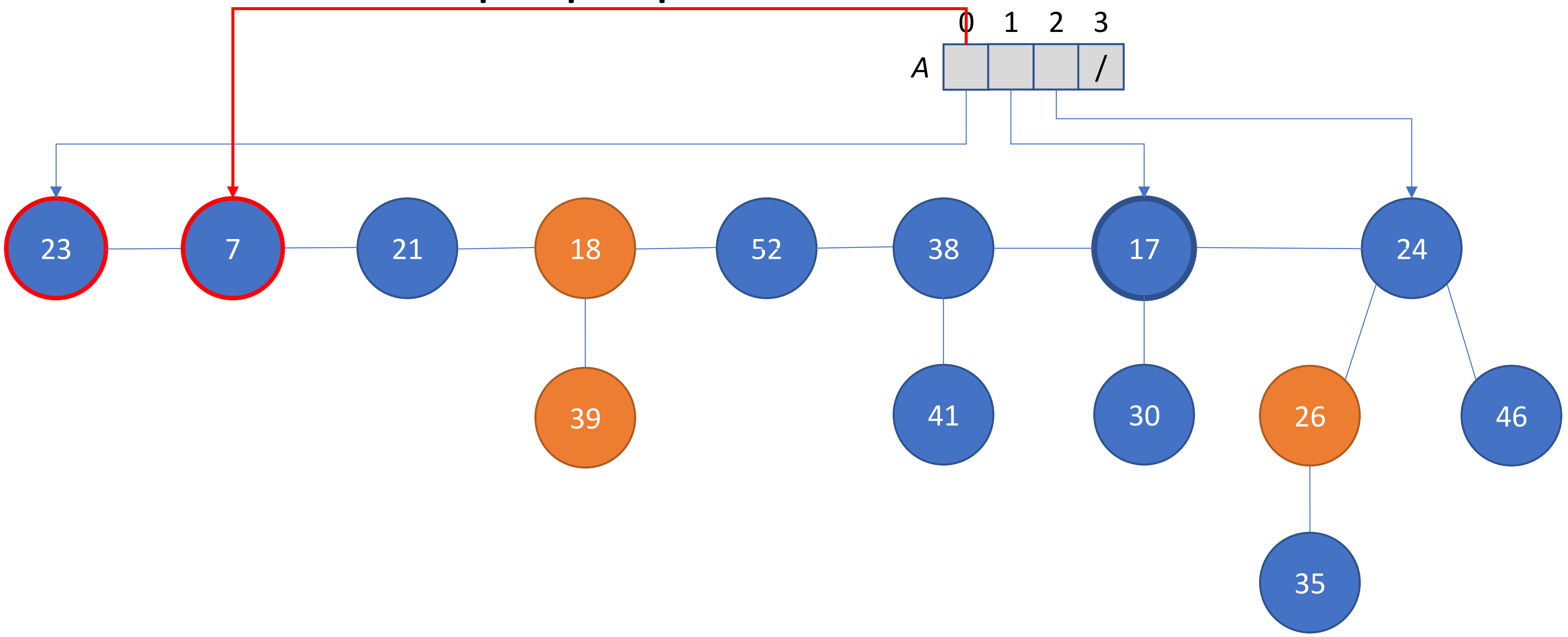
Fibonacci heap: pop min



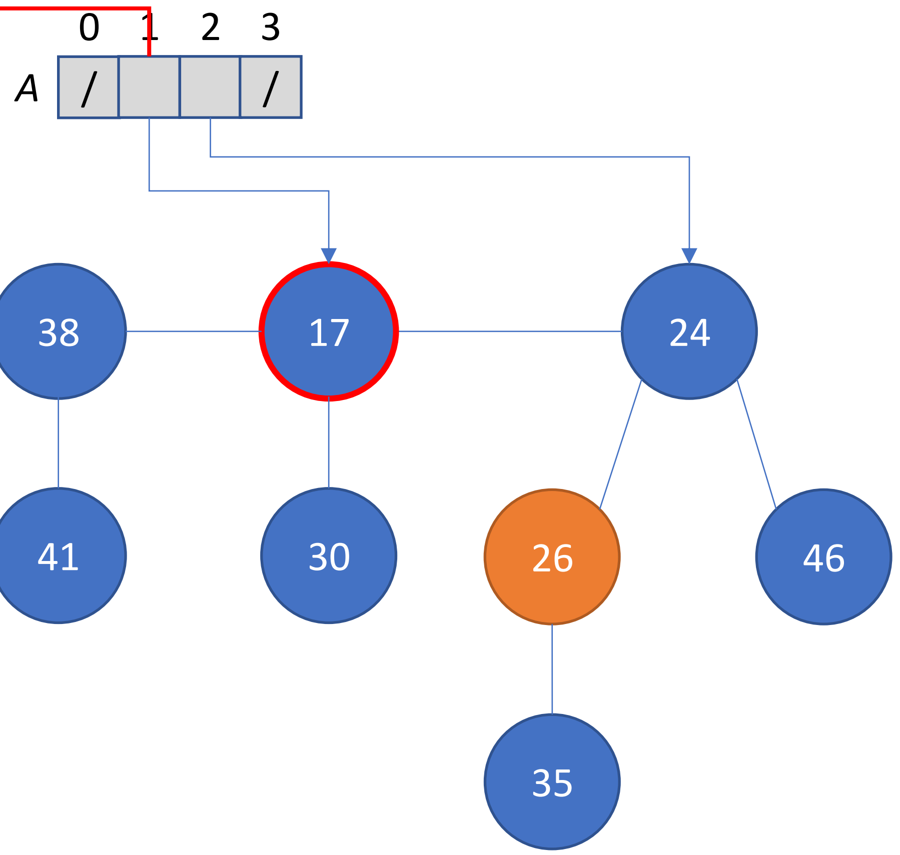
Fibonacci heap: pop min



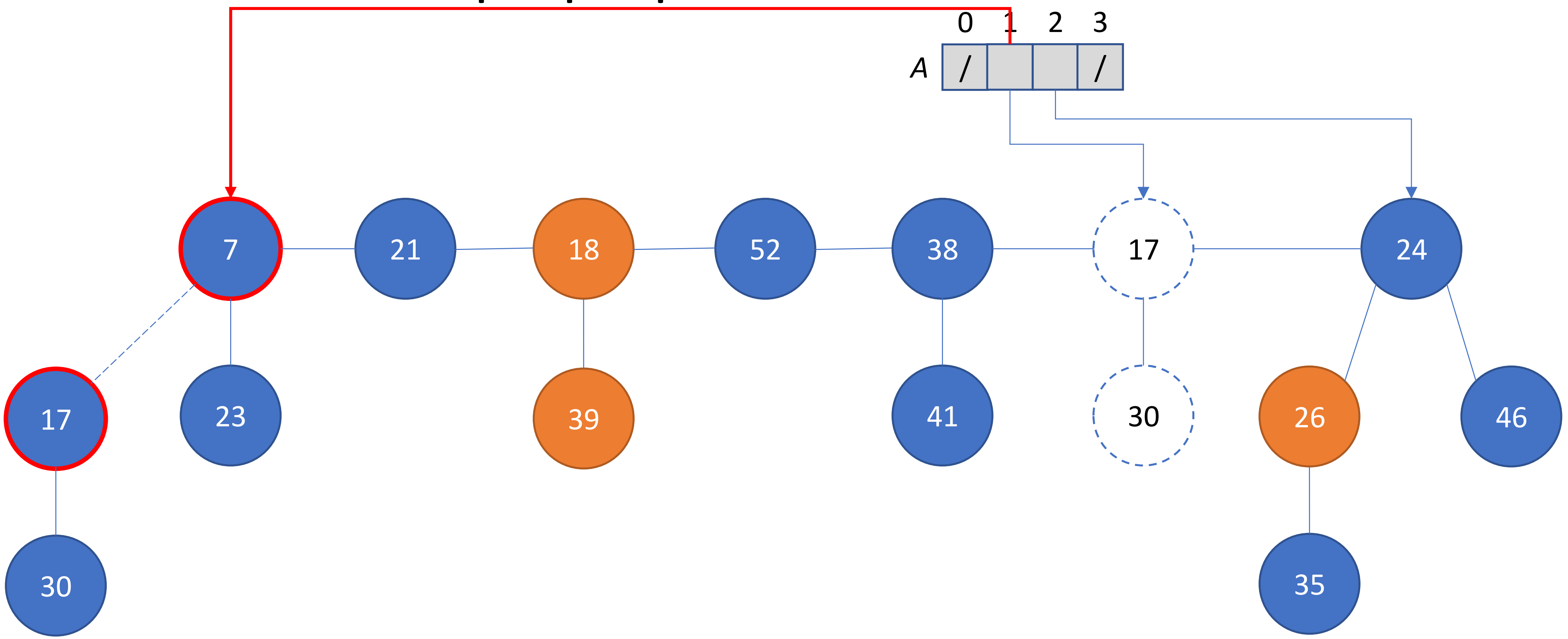
Fibonacci heap: pop min



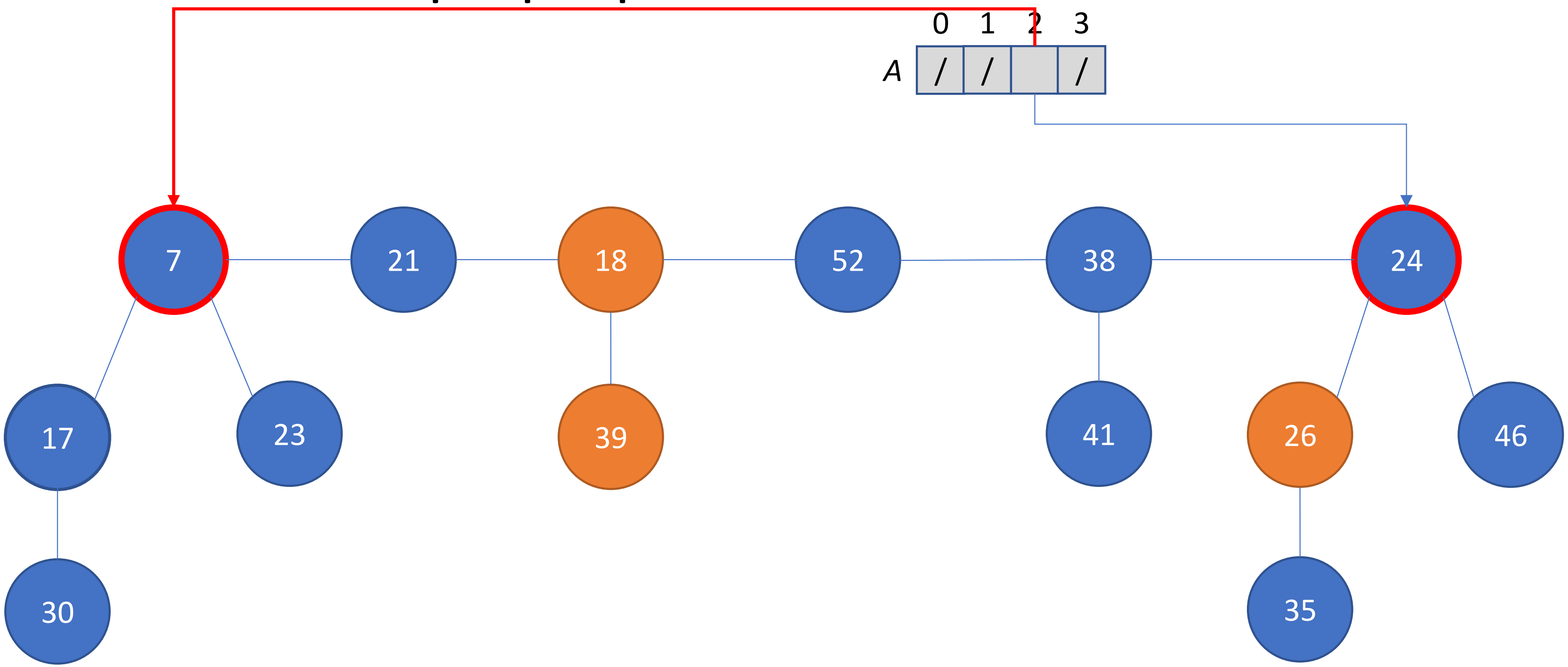
Fibonacci heap: pop min



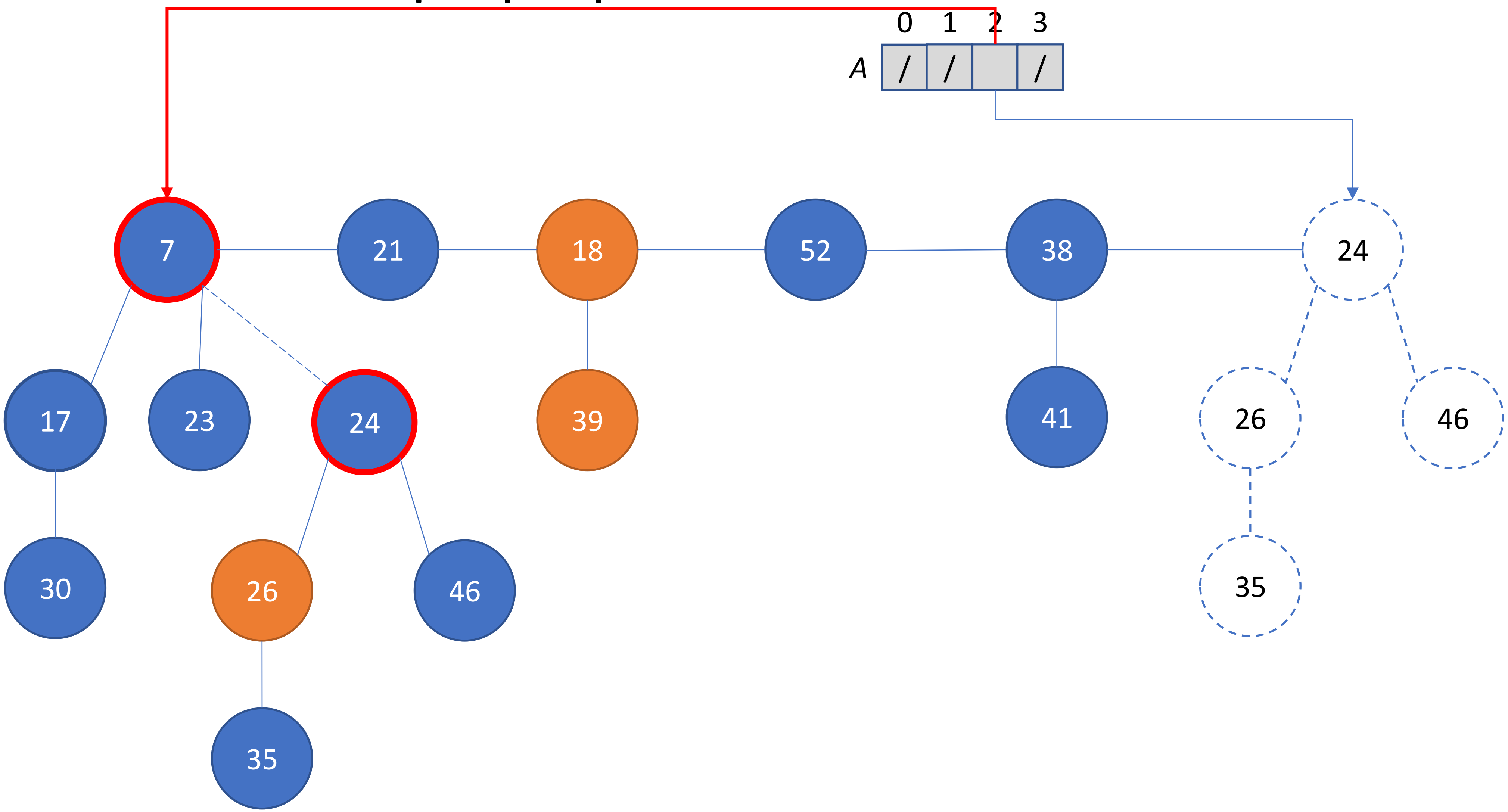
Fibonacci heap: pop min



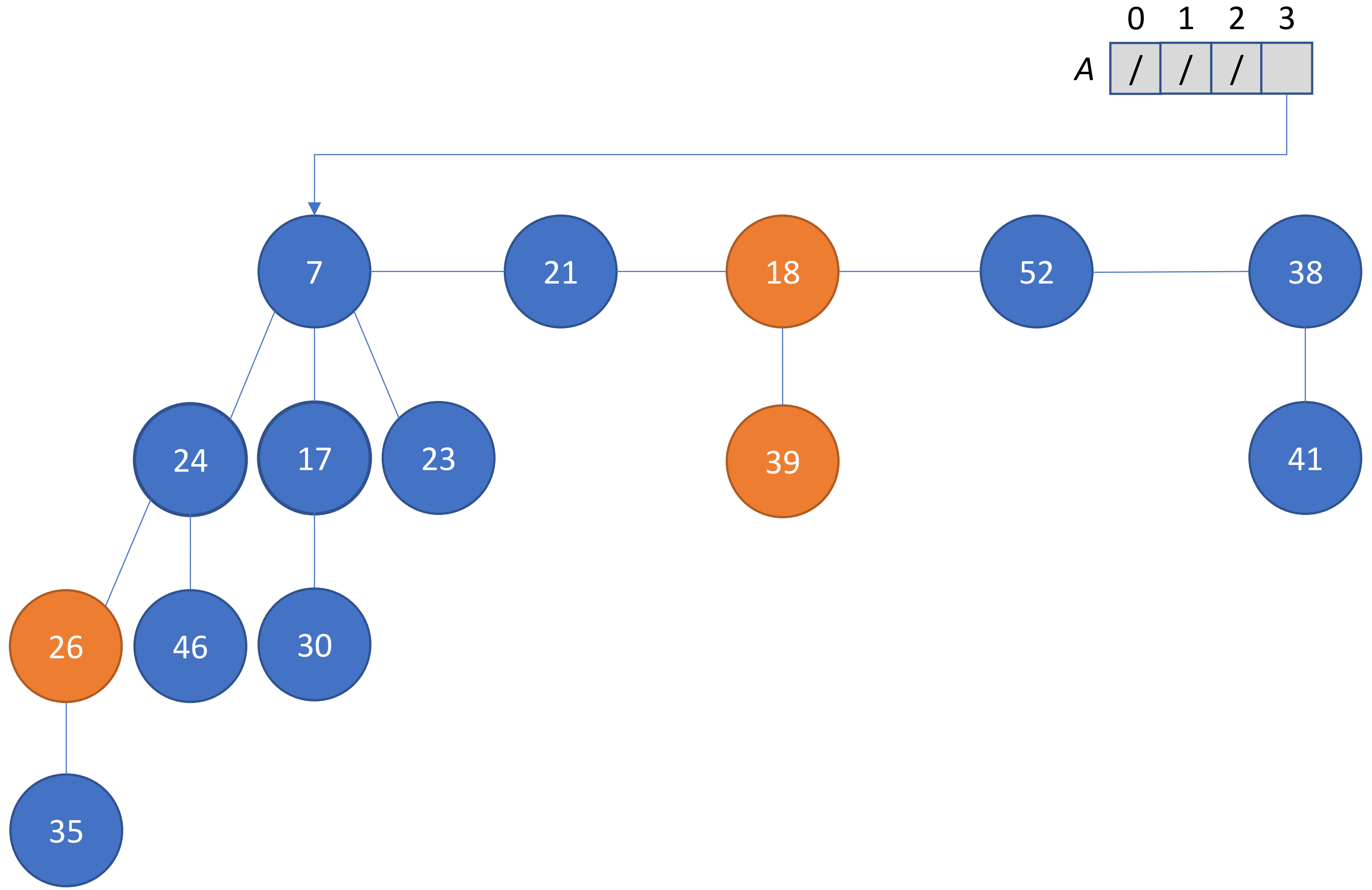
Fibonacci heap: pop min



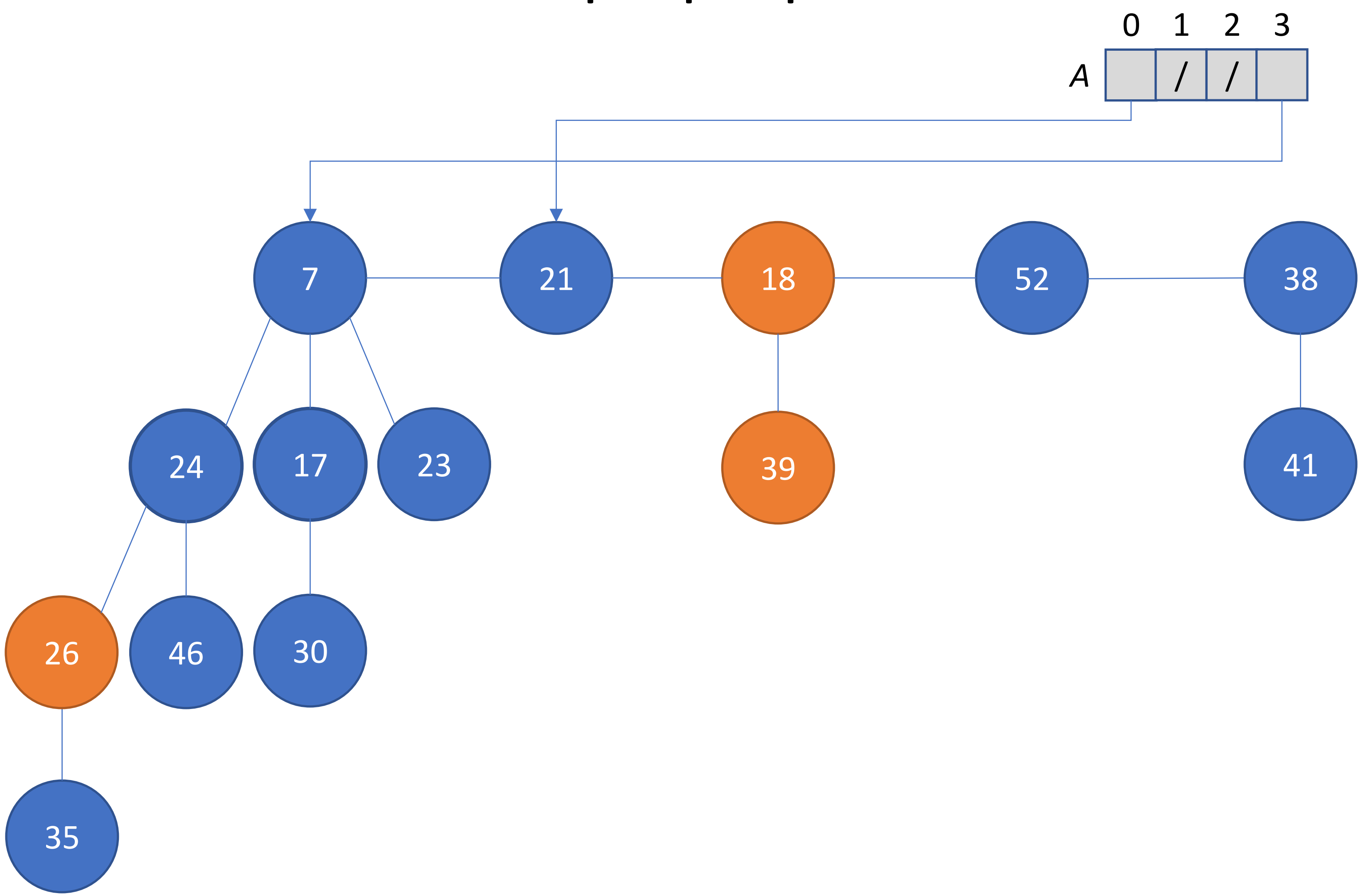
Fibonacci heap: pop min



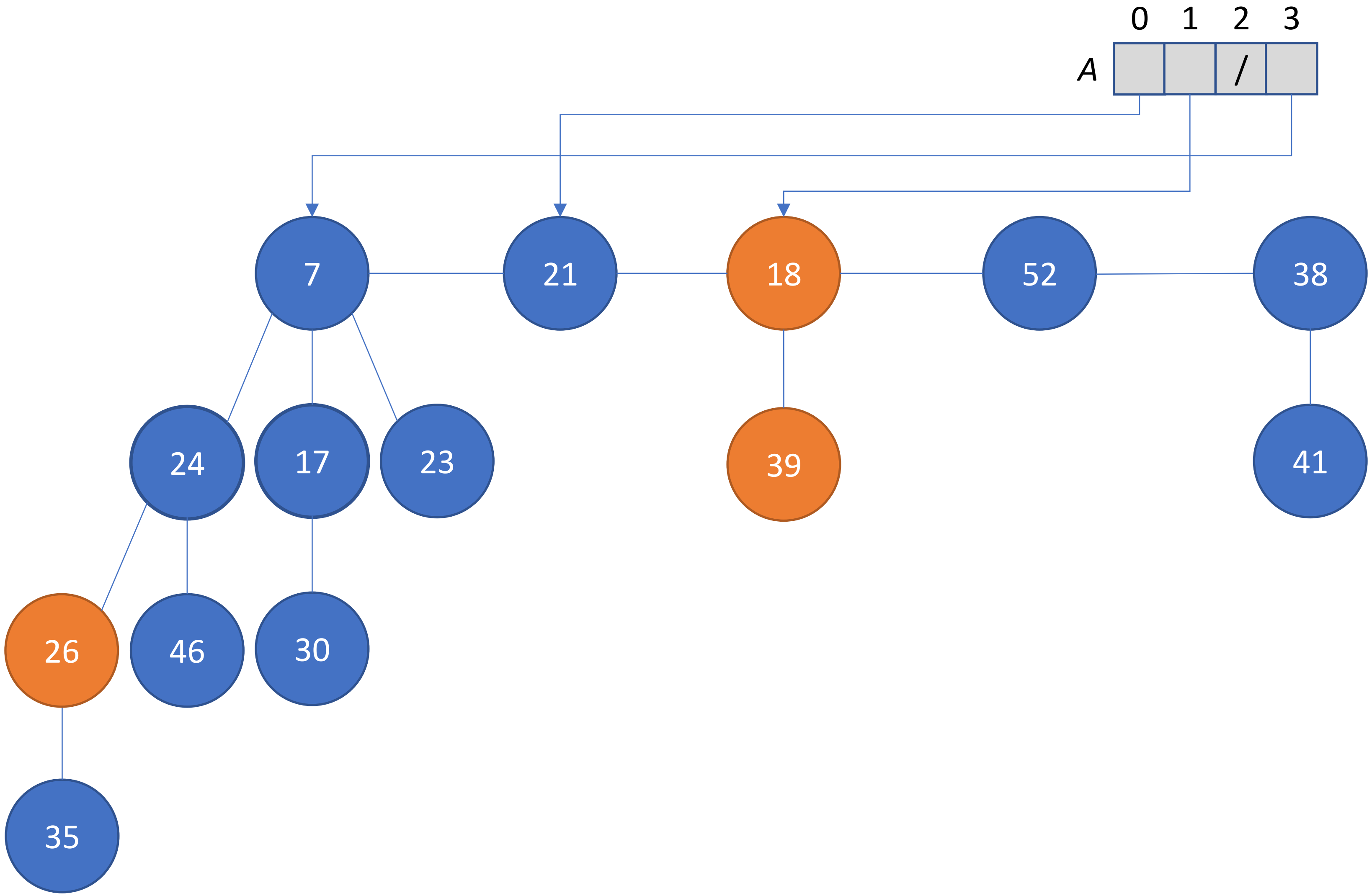
Fibonacci heap: pop min



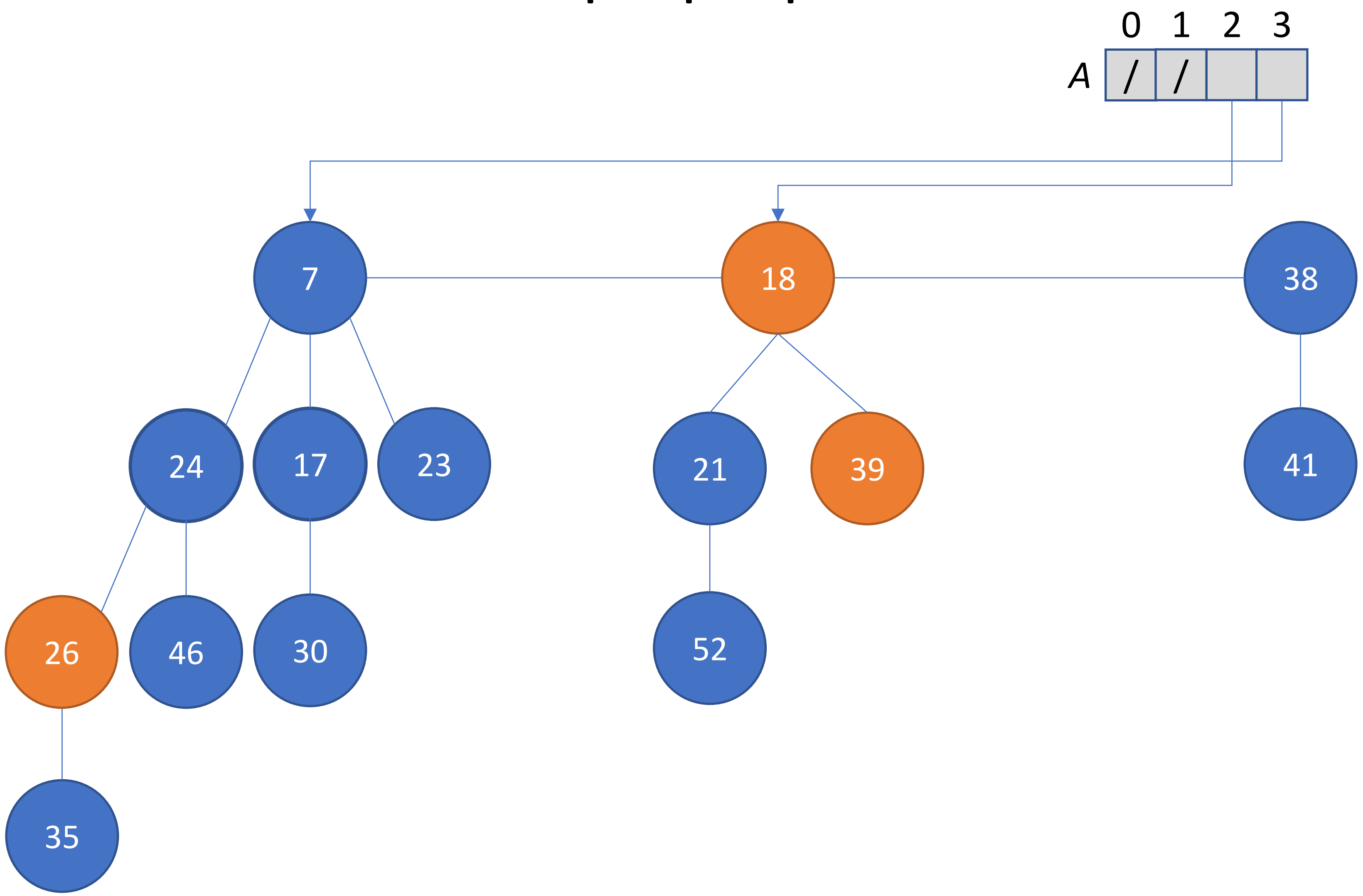
Fibonacci heap: pop min



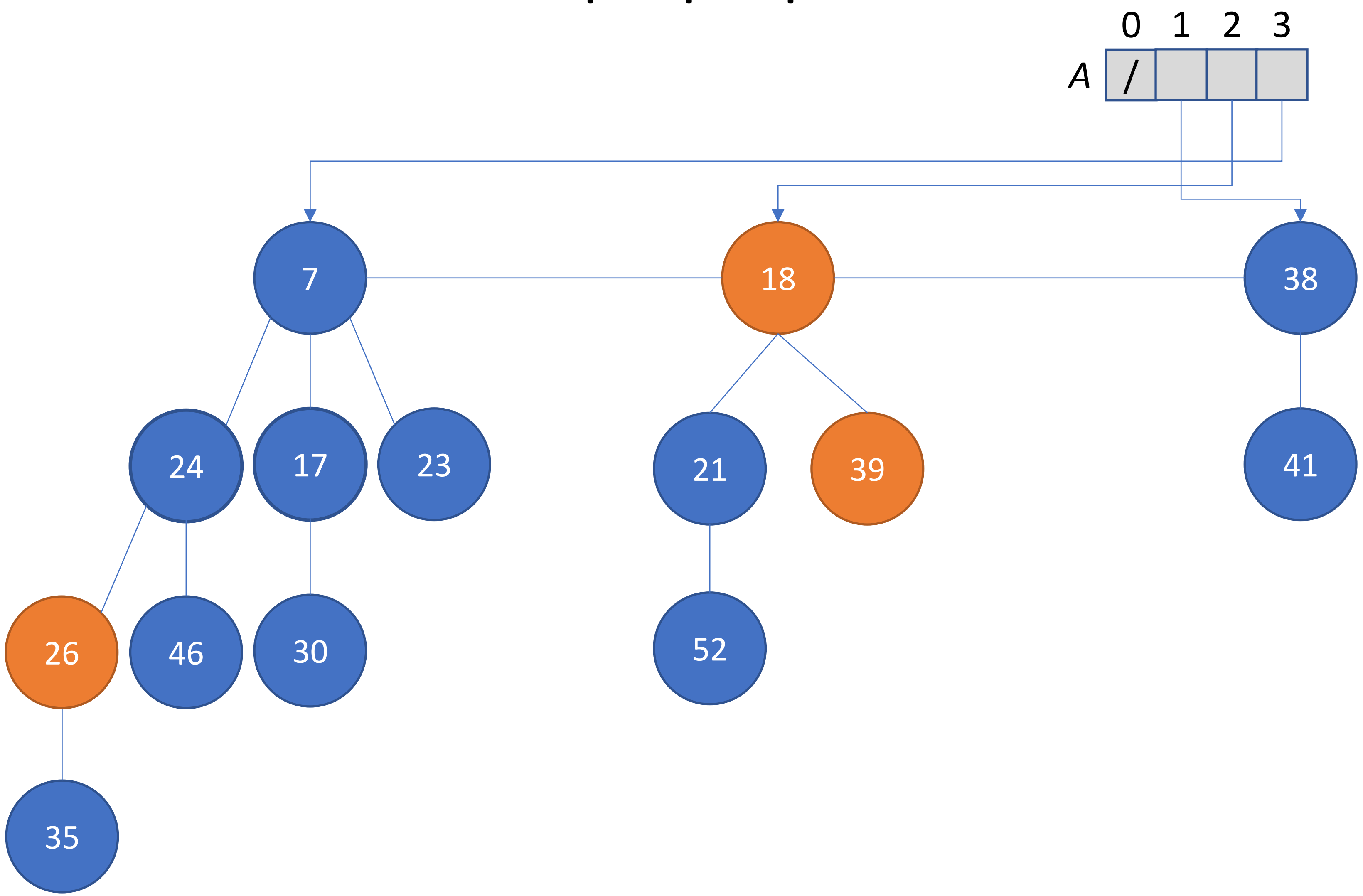
Fibonacci heap: pop min



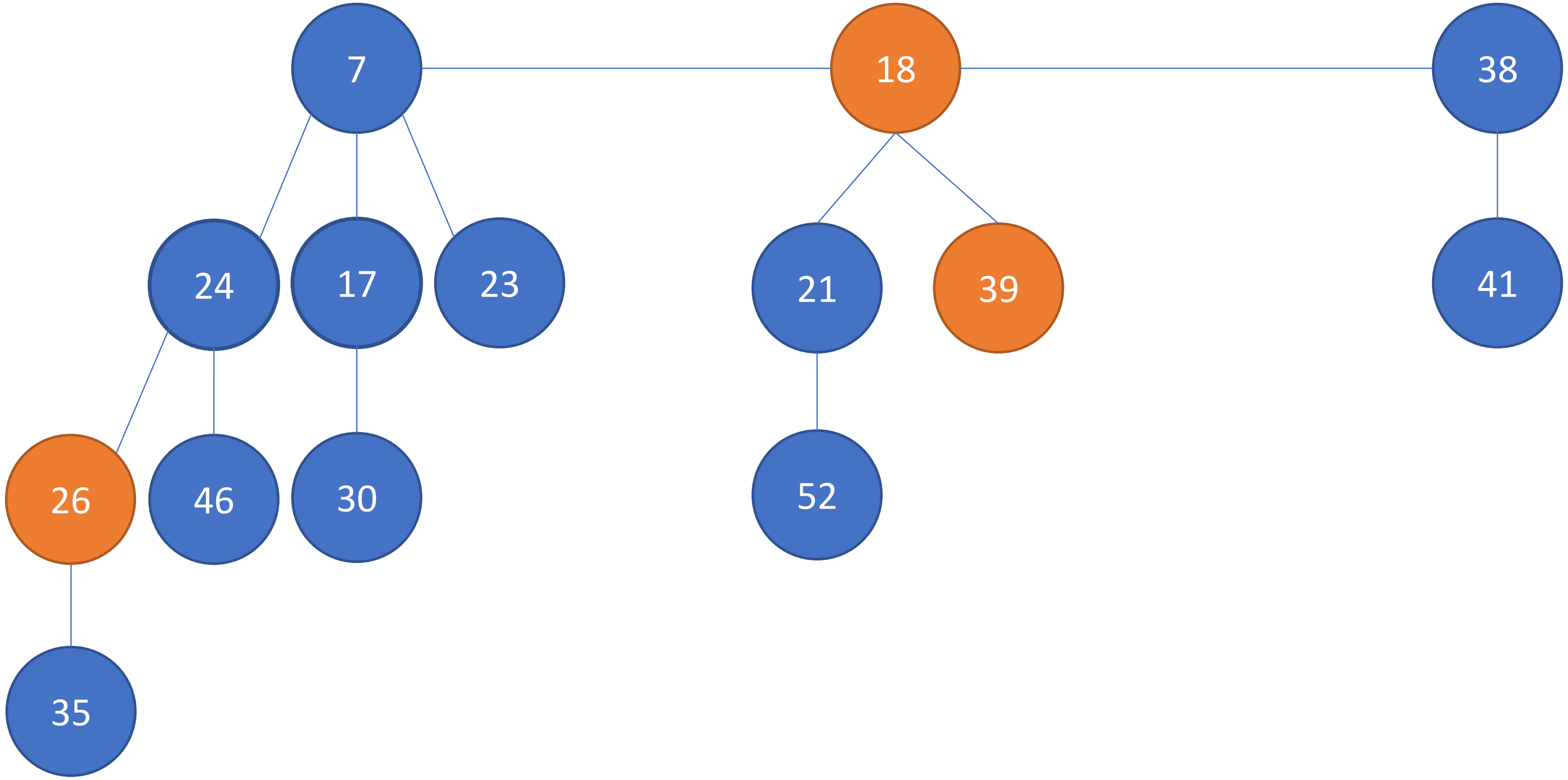
Fibonacci heap: pop min



Fibonacci heap: pop min



Fibonacci heap: pop min

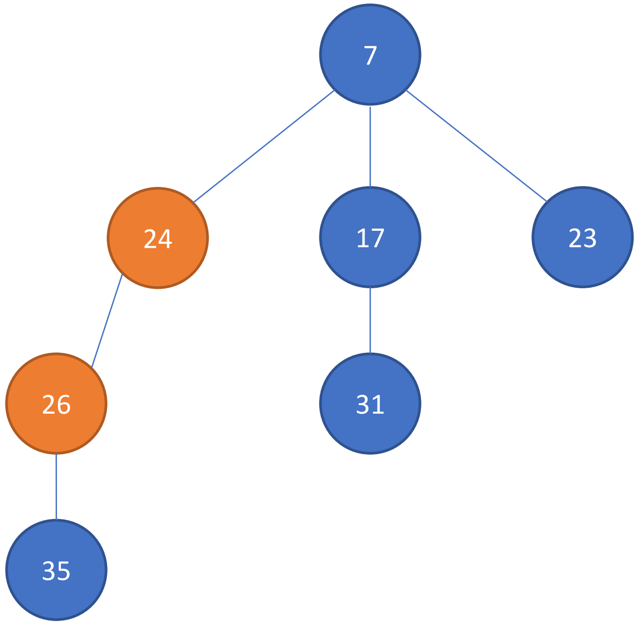


Fibonacci heap: decrease key

Idea:

- If the change did not break heap property – no structural changes required
- Otherwise, we cut out the node from its parent and make it a root node (mark parent, unmark node)
- If parent was marked – do cascade cut

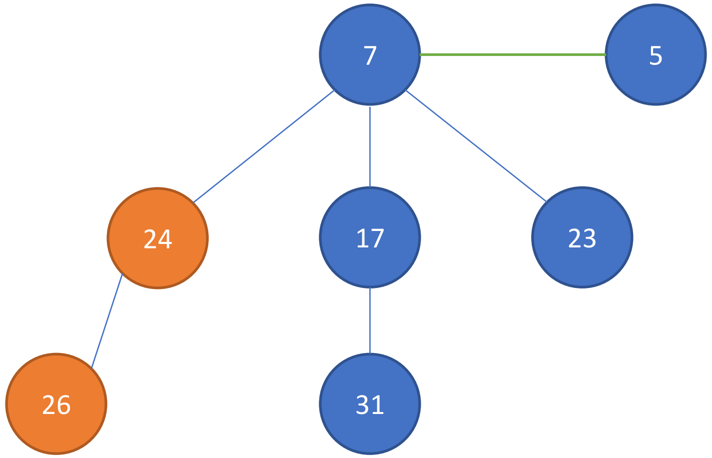
35 -> 5



Fibonacci heap: decrease key

Idea:

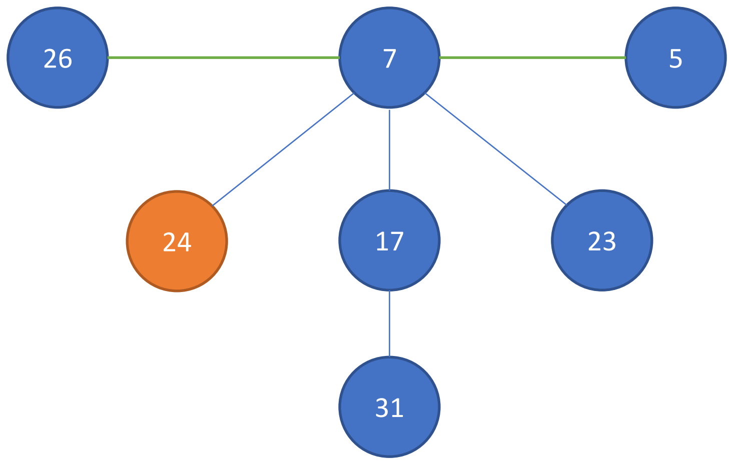
- If the change did not break heap property – no structural changes required
- Otherwise, we cut out the node from its parent and make it a root node (mark parent, unmark node)
- If parent was marked – do cascade cut



Fibonacci heap: decrease key

Idea:

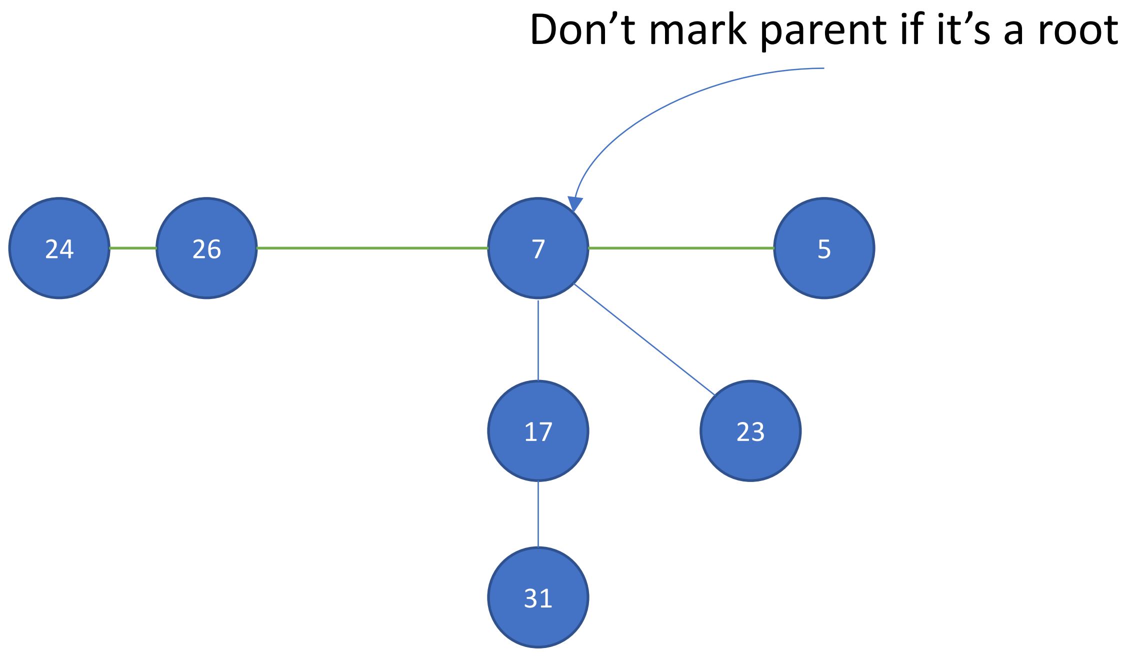
- If the change did not break heap property – no structural changes required
- Otherwise, we cut out the node from its parent and make it a root node (mark parent, unmark node)
- If parent was marked – do cascade cut



Fibonacci heap: decrease key

Idea:

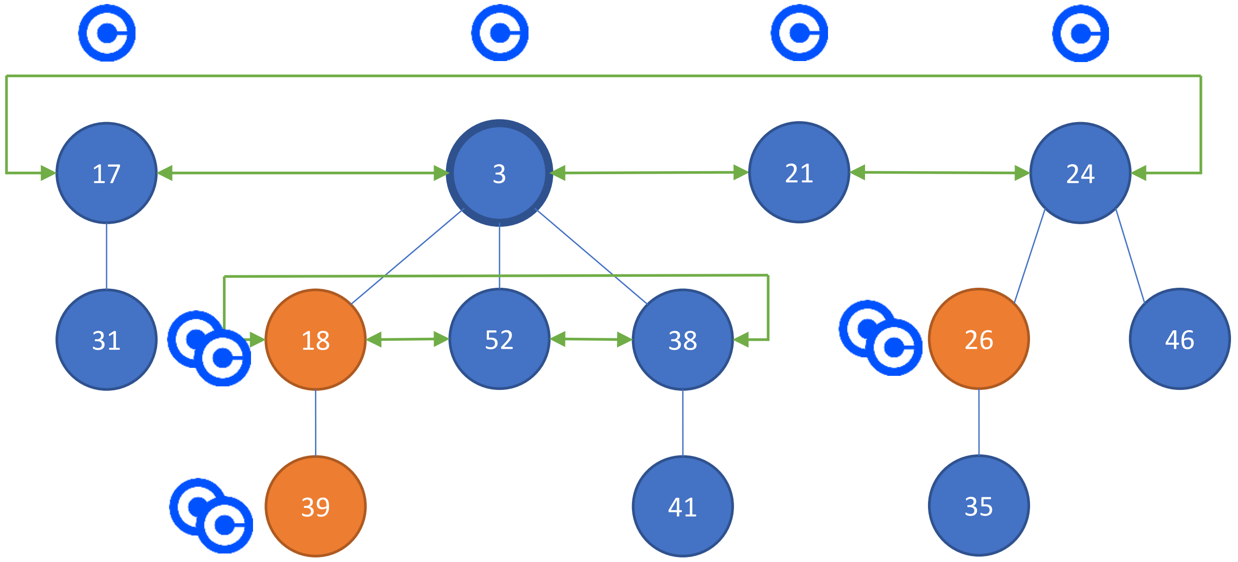
- If the change did not break heap property – no structural changes required
- Otherwise, we cut out the node from its parent and make it a root node (mark parent, unmark node)
- If parent was marked – do cascade cut

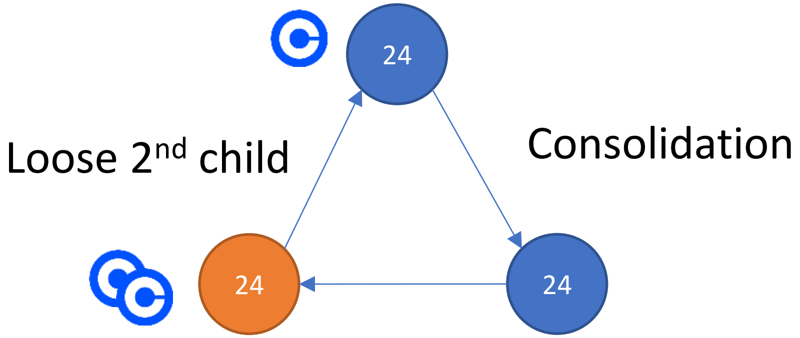


Fibonacci heap: amortized analysis

- Potential method:

$$\boldsymbol{\Phi(h) = trees(h) + 2 * marks(h) = t(h) + 2m(h)}.\ \Phi(h_0) = 0.$$





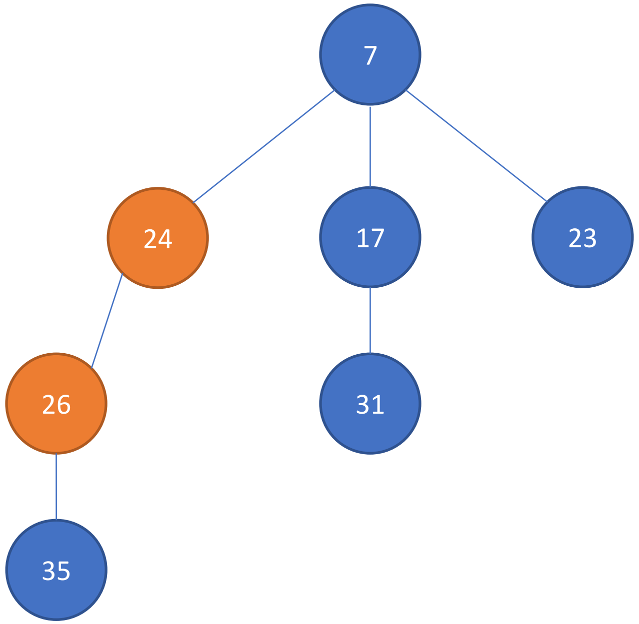
Fibonacci heap: amortized analysis

- Insert
- Actual cost $\Theta(1)$
- Potential change: $\Phi(h) = t(h) + 2m(h), \Delta\Phi = 1$
- Trivial.

Fibonacci heap: amortized analysis

- Decrease key
- Actual cost: $\Theta(?)$

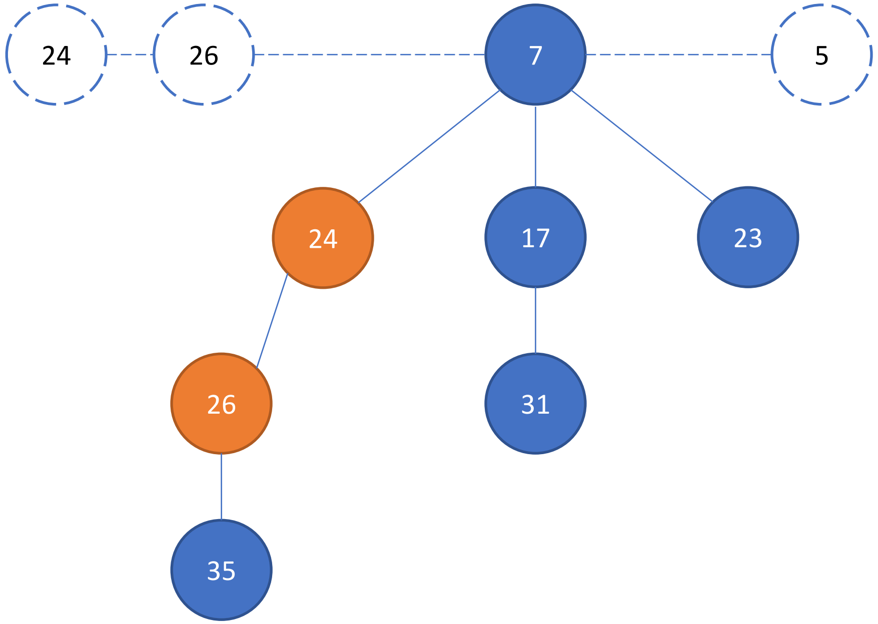
35 -> 5



Fibonacci heap: amortized analysis

- Decrease key
- Actual cost: $\Theta(cuts)$
- $t(h') = t(h) + cuts$

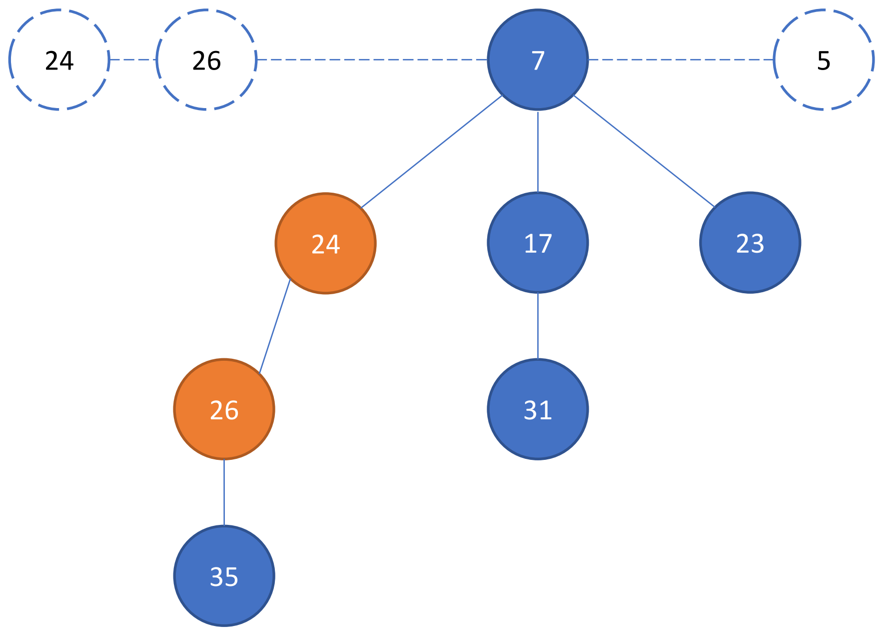
35 -> 5



Fibonacci heap: amortized analysis

- Decrease key
- Actual cost: $\Theta(cuts)$
- $t(h') = t(h) + cuts$
- $m(h') \leq m(h) - cuts + 2$

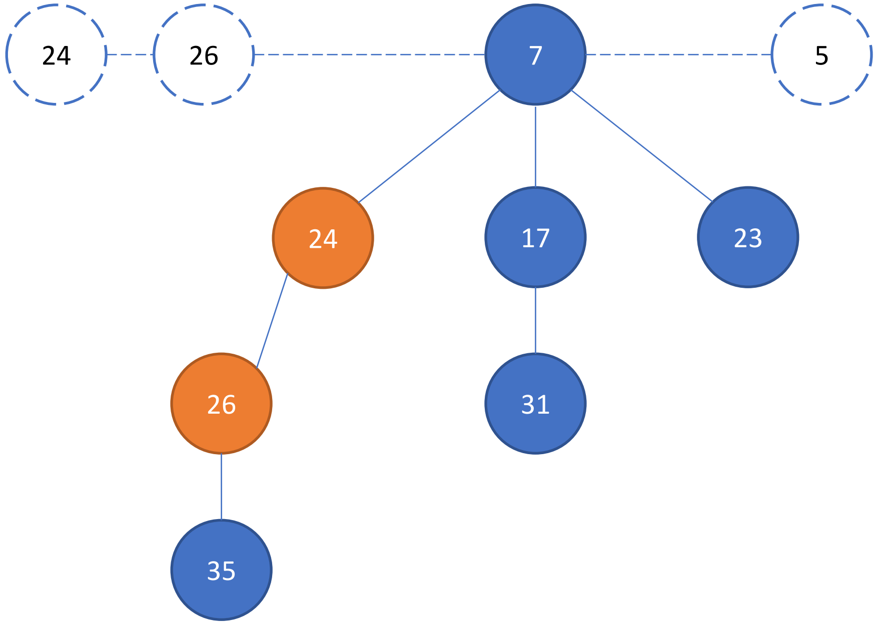
35 -> 5



Fibonacci heap: amortized analysis

- Decrease key
- Actual cost: $\Theta(cuts)$
- $t(h') = t(h) + cuts$
- $m(h') \leq m(h) - cuts + 2$
- $\Delta\Phi \leq cuts + 2(-cust + 2) = 4 - cuts = \Theta(1) - cuts$
- Amortized cost: $\Theta(1)$

35 -> 5

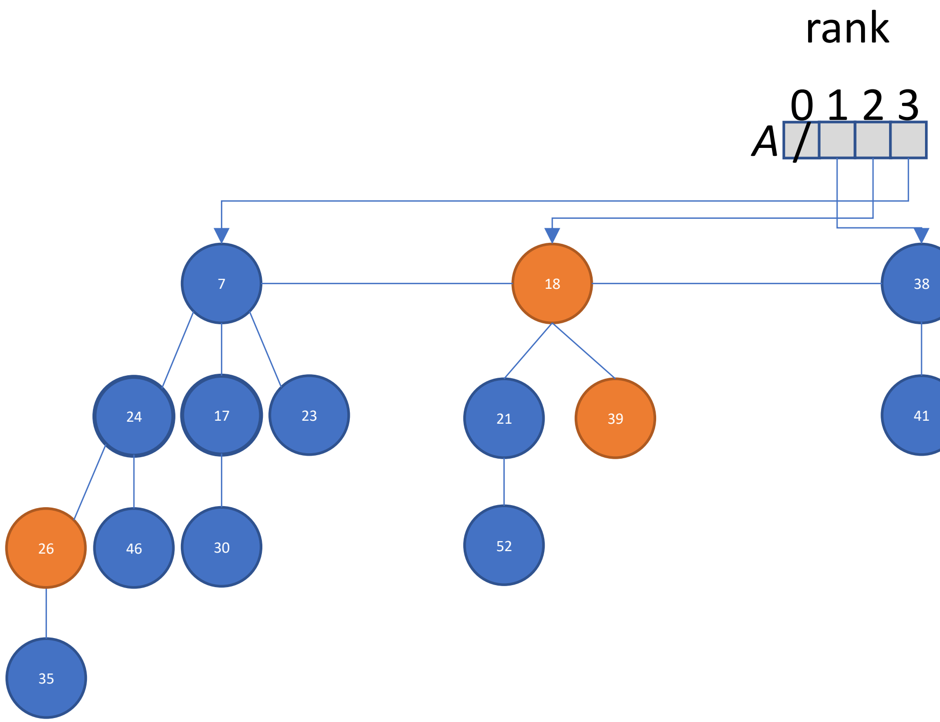


$$\boldsymbol{\Phi(h) = t(h) + 2m(h)}$$

1 – for paying the cut, 1 – for $t(h)$ increase

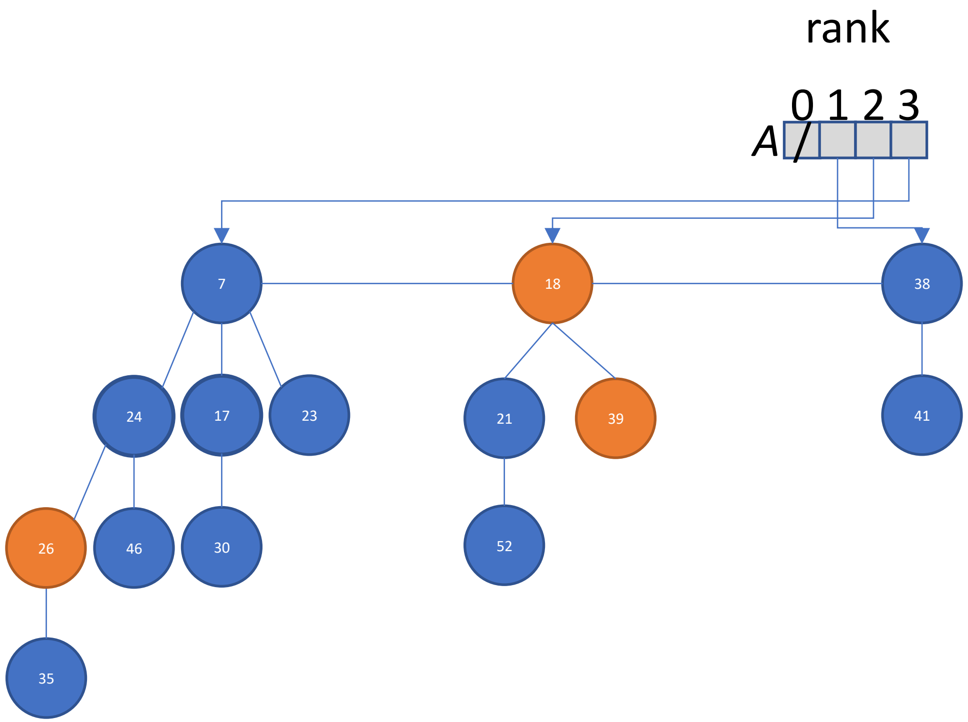
Fibonacci heap: amortized analysis

- Pop min
- Actual cost: ?



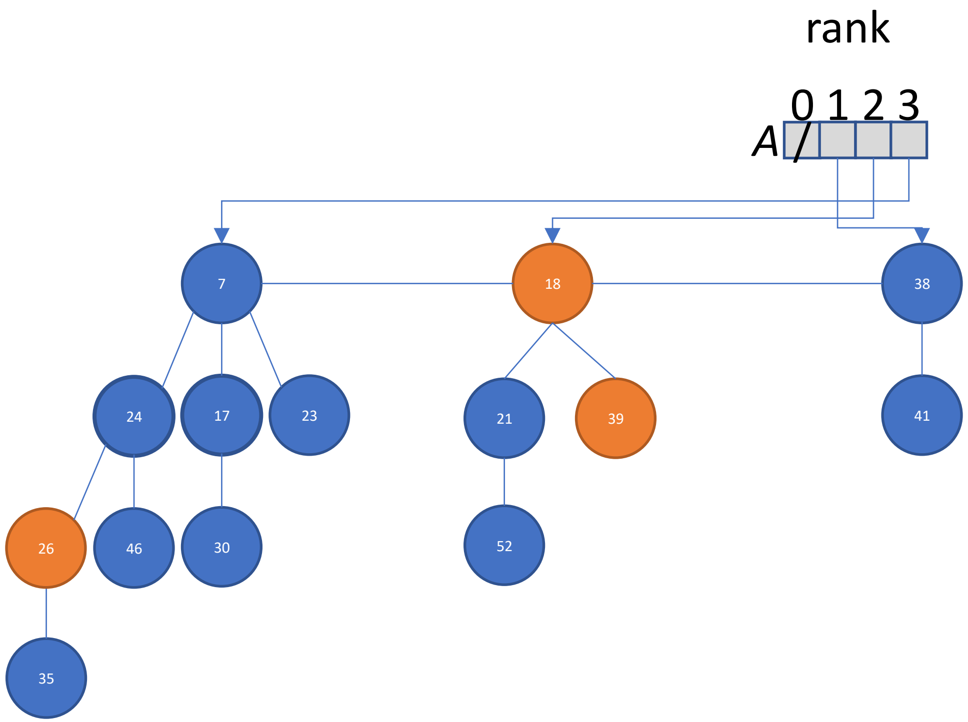
Fibonacci heap: amortized analysis

- Pop min
- Actual cost:
  - Meld min's children into root
  - Update min
  - Consolidate trees



Fibonacci heap: amortized analysis

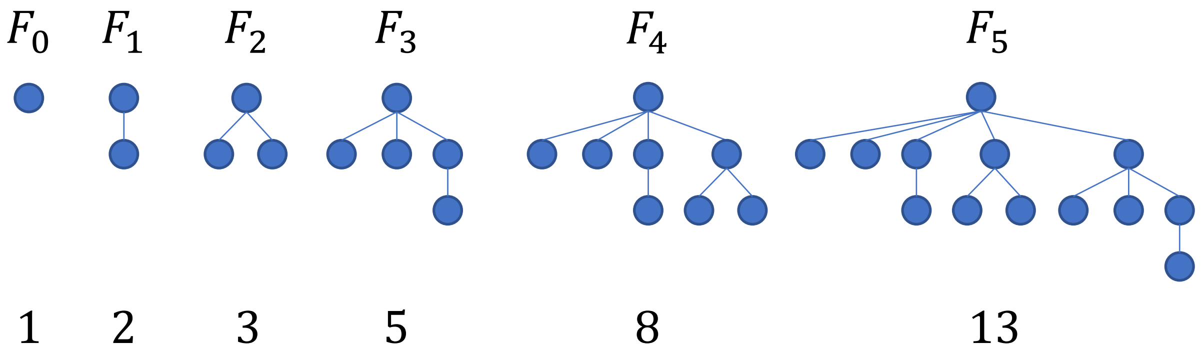
- Pop min
- Actual cost:
  - Meld min's children into root
    - $\Theta(rank(h))$
  - Update min
    - $\Theta(rank(h)) + \Theta(t(h))$
  - Consolidate trees
    - $\Theta(rank(h)) + \Theta(t(h))$
- Potential change:
  - $t(h') \leq rank(h) + 1$
  - $\Delta\Phi = rank(h) + 1 - t(h)$



- Amortized cost: $\Theta(rank(h))$

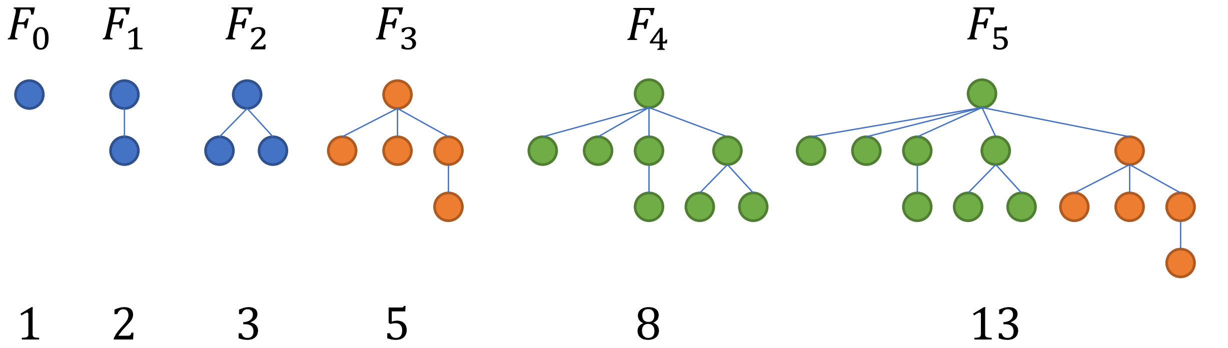
Fibonacci heap: amortized analysis

- Pop min
- $\Theta(rank(h))$



Fibonacci heap: amortized analysis

- Pop min
- $\Theta(rank(h))$
- $F_k \geq \varphi^k, \varphi = (1 + \sqrt{5})/2$
- $rank(h) \leq \log_\varphi n$



Amortized cost: $\Theta(rank(h)) = \Theta(\log(n))$

Summary

- Fibonacci heap

Method	complexity
Insert	$\Theta(1)$
Get min	$\Theta(1)$
Pop min	$\Theta(\log n)^+$
Decrease key	$\Theta(1)^+$
Merge*	$\Theta(1)$

- Ordinary heap

Method	complexity
Insert	$\Theta(\log n)$
Get max	$\Theta(1)$
Pop max	$\Theta(\log n)$
Increase key	$\Theta(\log n)$
Merge*	$\Theta(n)$

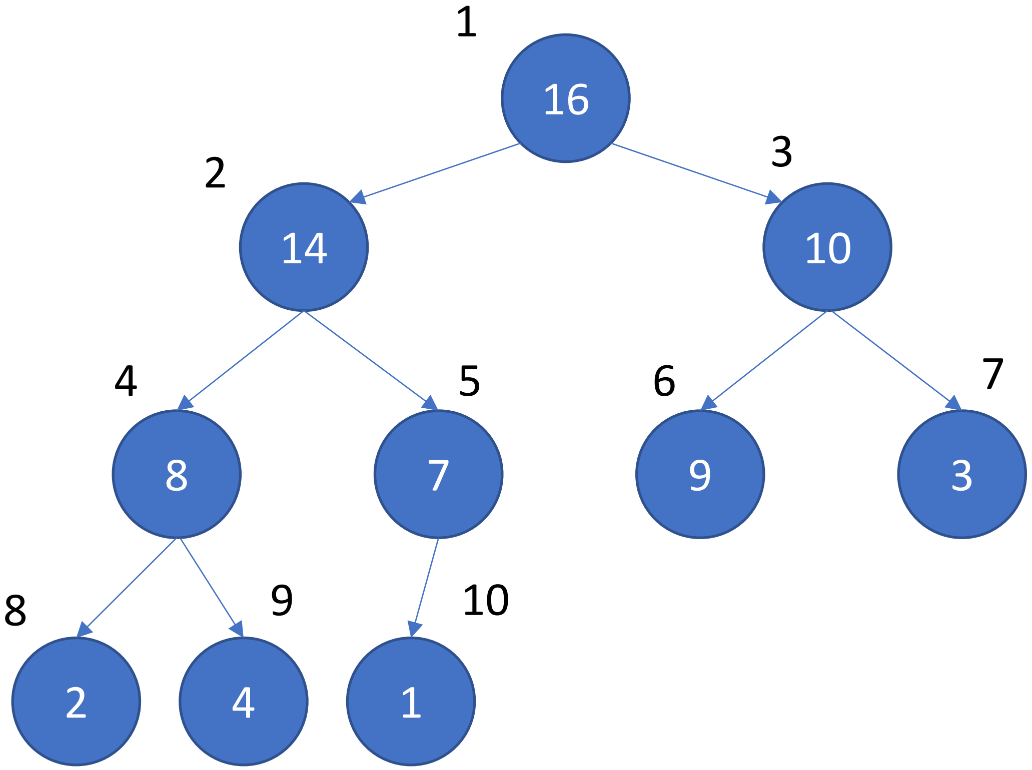
Resources

- Introduction to Algorithms, Thomas H. Cormen, chapters 17,19.
- Classic: https://link.springer.com/content/pdf/10.1007/BF01683268.pdf
- Comparative study for parallel and sequential priority queues: https://dl.acm.org/doi/pdf/10.1145/249204.249205

BACKUP

Heap: insertion

- Max-Heap:



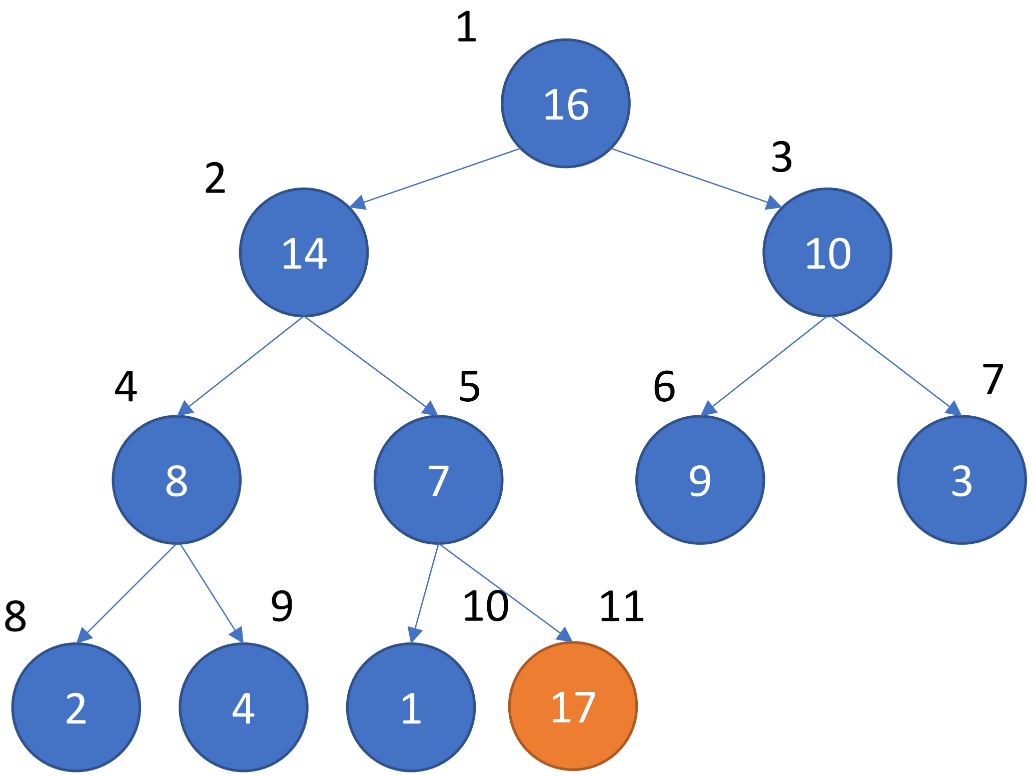
Complete* binary tree

New node

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Heap: insertion

- Max-Heap:



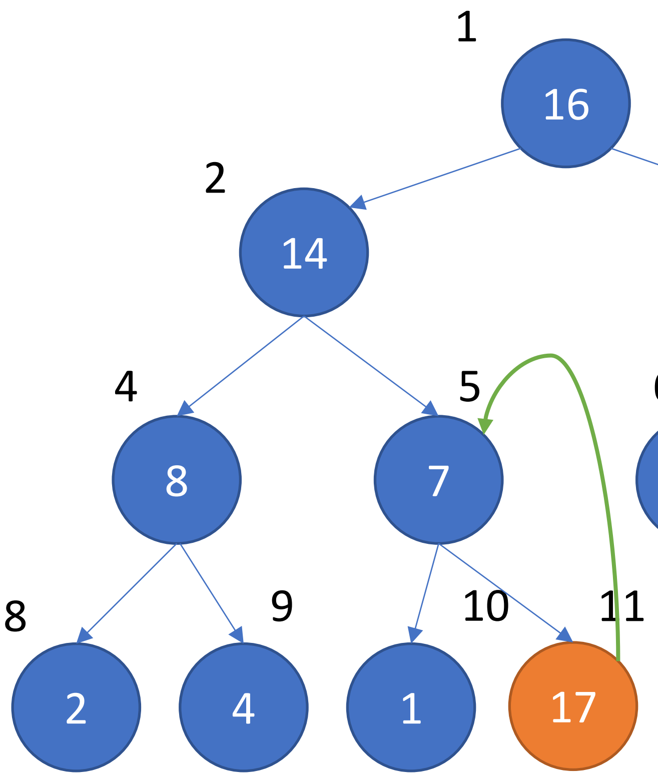
Complete* binary tree

- Push back

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	17

Heap: insertion

- Max-Heap:



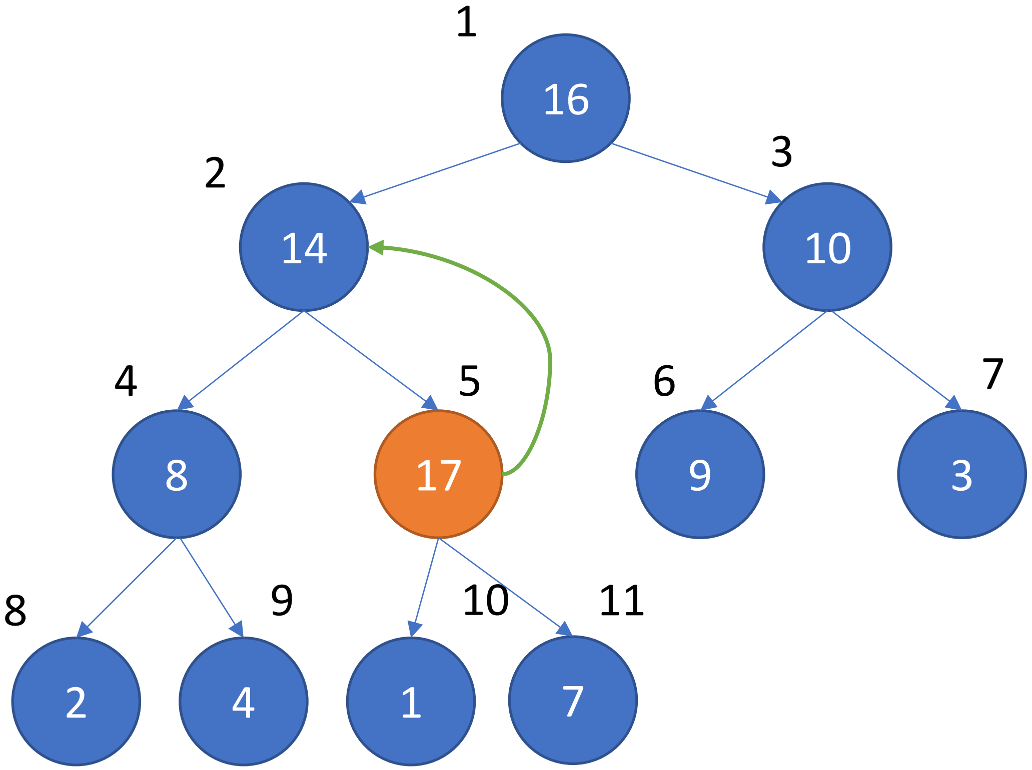
Complete* binary tree

- Push back
- Bubble it up, comparing to parent

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	17

Heap: insertion

- Max-Heap:



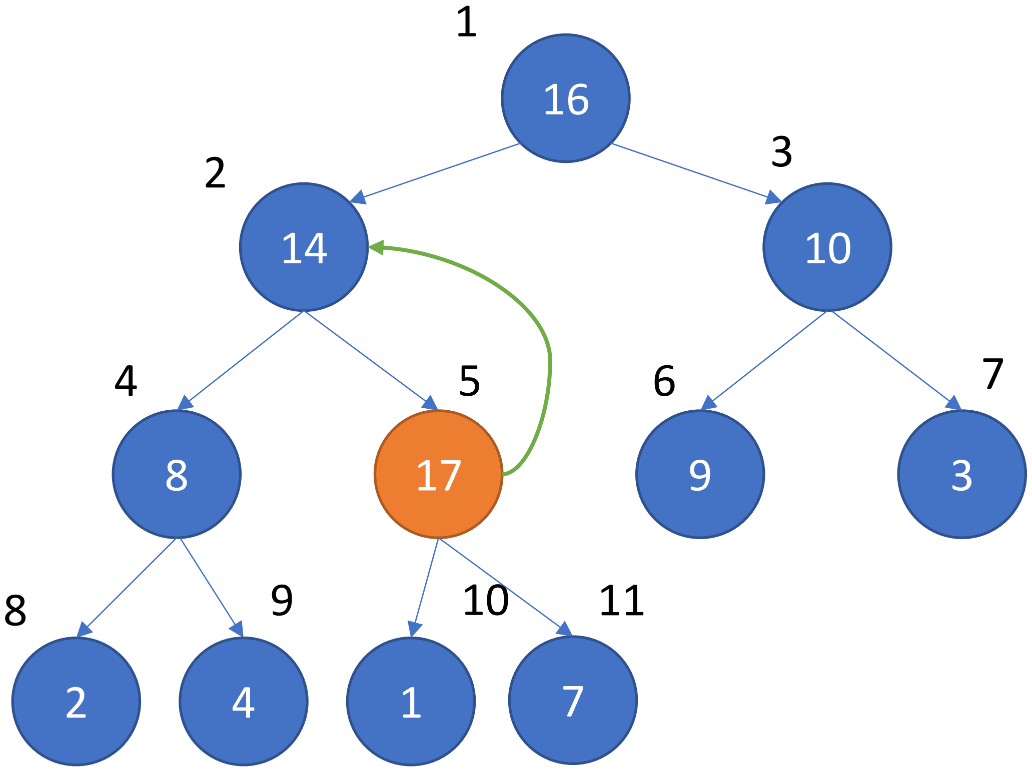
Complete* binary tree

- Push back
- Bubble it up, comparing to parent
- Until in place
- Complexity?

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	17	9	3	2	4	1	7

Heap: insertion

- Max-Heap:



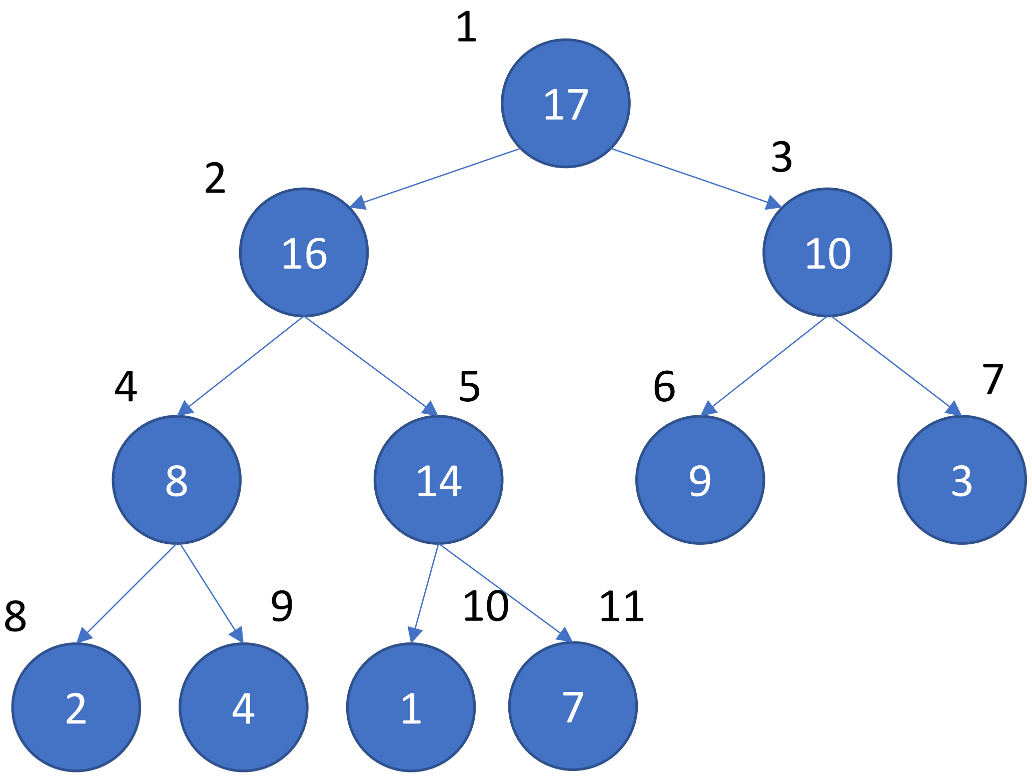
Complete* binary tree

- Push back
- Bubble it up, comparing to parent
- Until in place
- Complexity: $\Theta(\log n)$

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	17	9	3	2	4	1	7

Heap: pop

- Max-Heap:



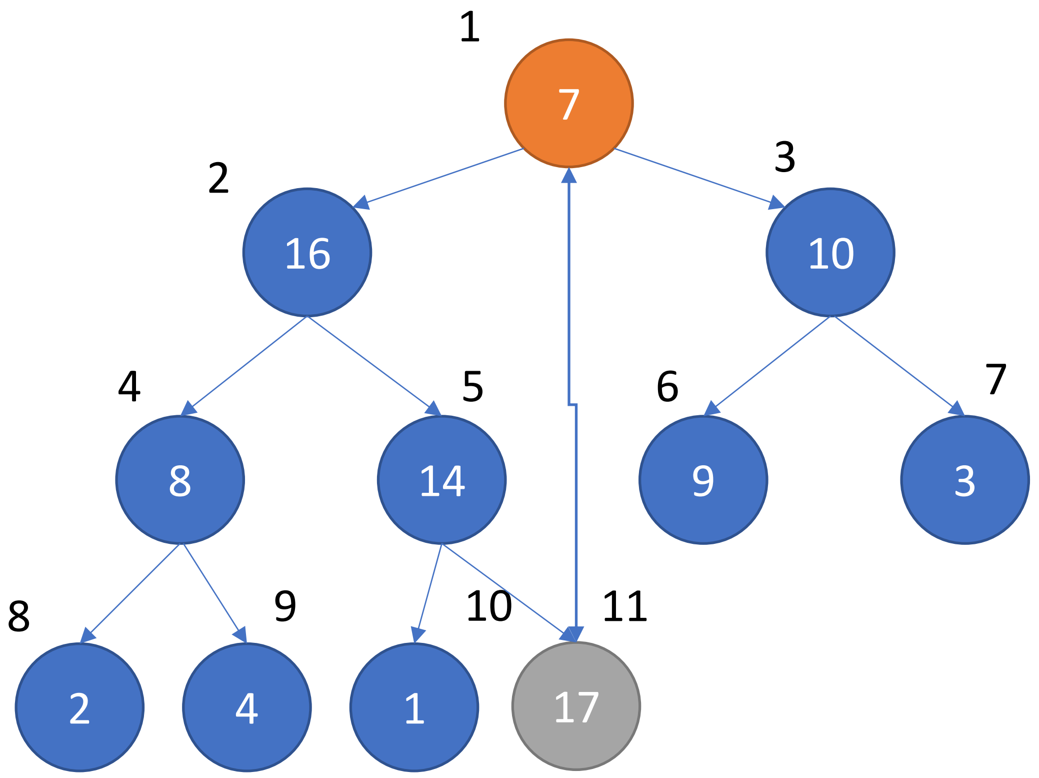
Complete* binary tree

- Let’s pop largest (min) element

1	2	3	4	5	6	7	8	9	10	11
17	16	10	8	14	9	3	2	4	1	7

Heap: pop

- Max-Heap:



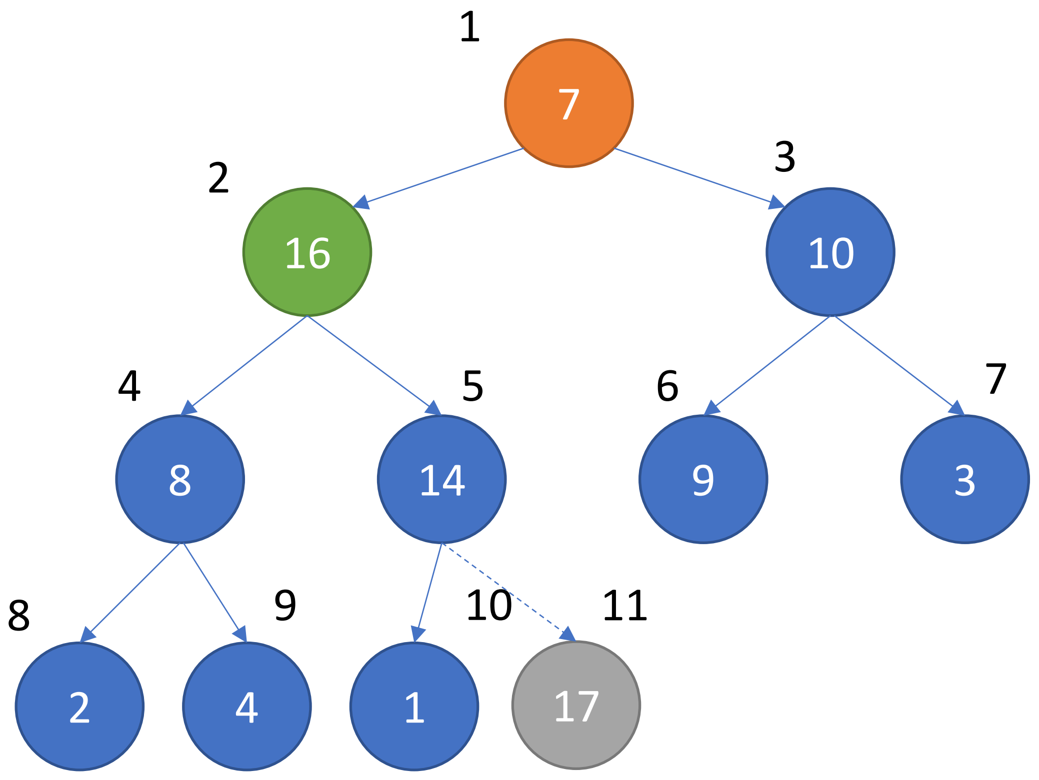
Complete* binary tree

- Let’s pop largest (min) element
- Swap last with root

1	2	3	4	5	6	7	8	9	10	11
7	16	10	8	14	9	3	2	4	1	17

Heap: pop

- Max-Heap:



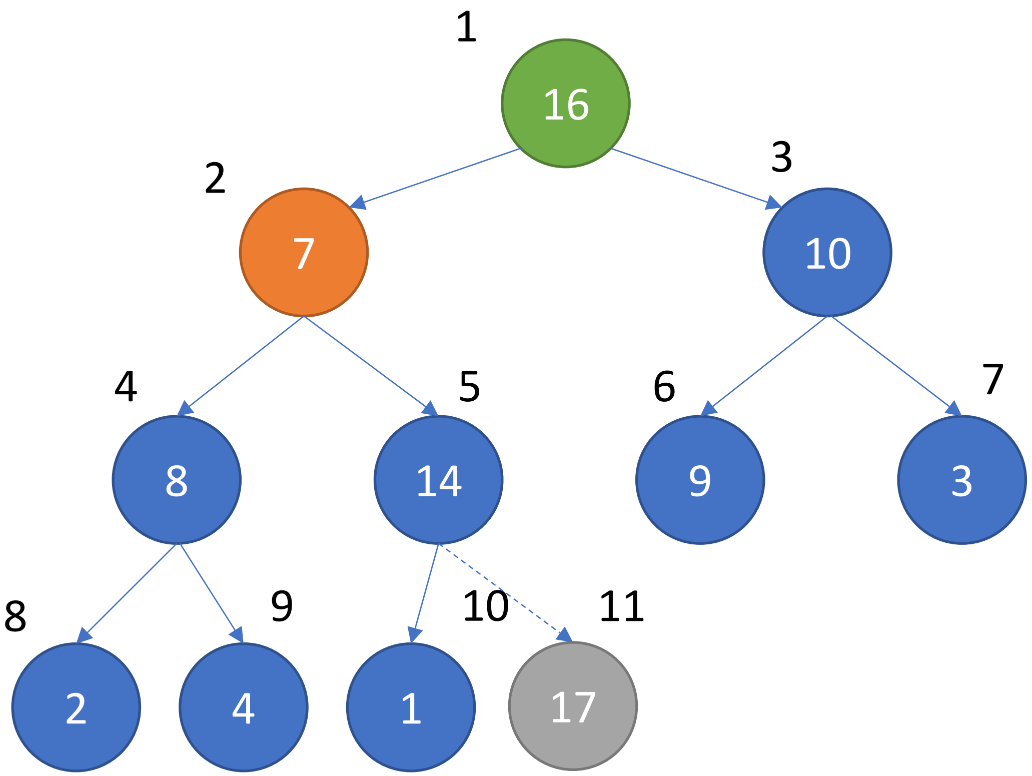
Complete* binary tree

- Let’s pop largest (min) element
- Swap last with root
- Push down swapping with largest child

1	2	3	4	5	6	7	8	9	10	11
7	**16**	10	8	14	9	3	2	4	1	17

Heap: pop

- Max-Heap:



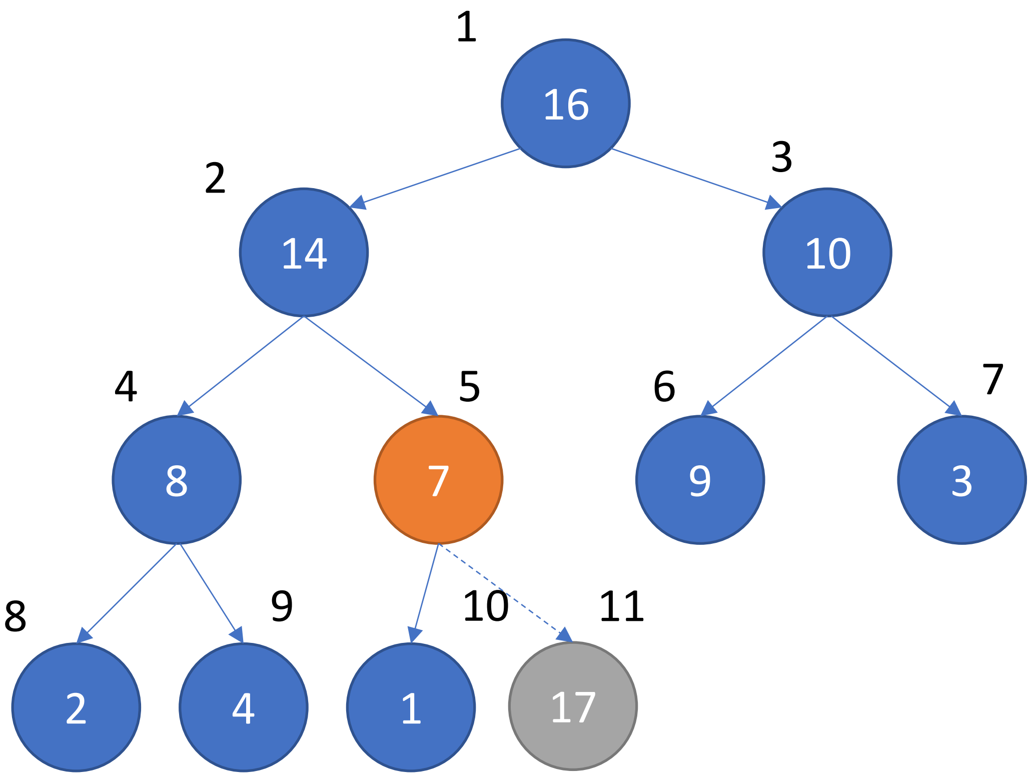
Complete* binary tree

- Let’s pop largest (min) element
- Swap last with root
- Push down swapping with largest child

1	2	3	4	5	6	7	8	9	10	11
16	7	10	8	14	9	3	2	4	1	17

Heap: pop

- Max-Heap:



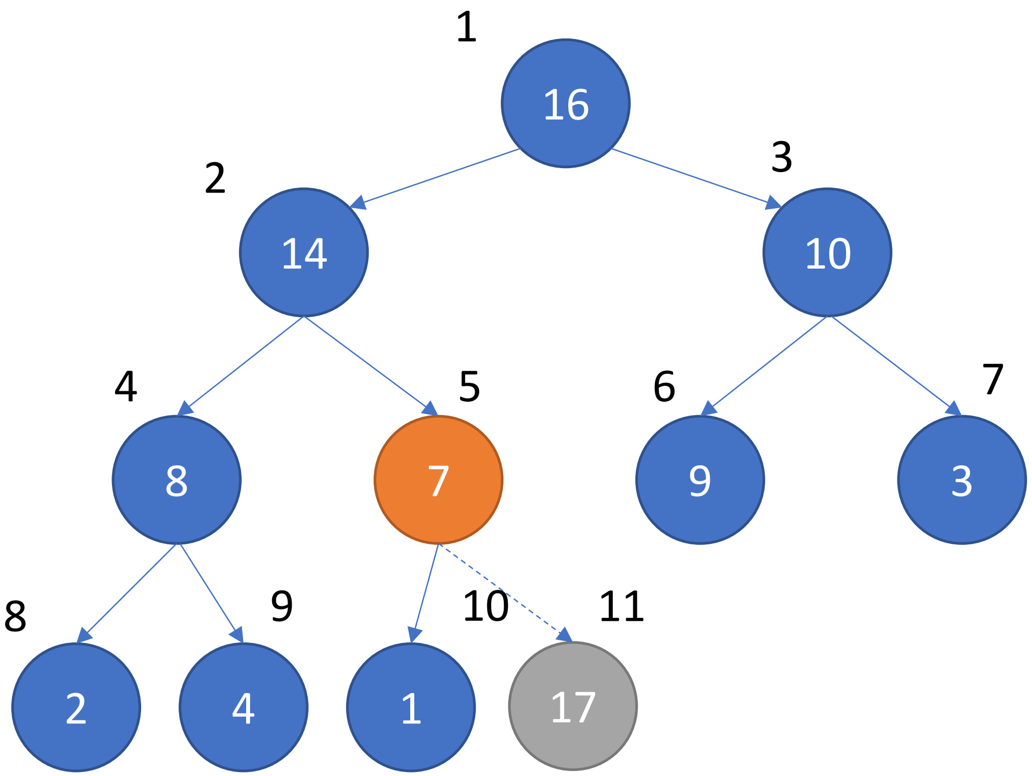
Complete* binary tree

- Let’s pop largest (min) element
- Swap last with root
- Push down swapping with largest child
- Complexity?

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	17

Heap: pop

- Max-Heap:



Complete* binary tree

- Let’s pop largest (min) element
- Swap last with root
- Push down swapping with largest child
- Complexity: $\Theta(\log n)$

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	17