Amortized Analysis

Thomas Schwarz, SJ

Amortized Analysis

- Determine the average time of a series of operations
  - Allows to optimize average performance
  - Example: Linear Hashing with fixed load factor α.
    - Cost of bucket split is divided over $\frac{1}{\alpha}$ inserts

Amortized Analysis

- Aggregate Analysis
  - Determine upper bound $T(n)$ on a sequence of n operations
  - Average cost is then $T(n)/n$
- Accounting method:
  - Most operations gets overcharged
  - Accumulated overcharges are used to pay for later operations
- Potential method:
  - Model the “credit” as a potential

Aggregate Analysis

- Stacks have
  - push
  - pop
  - empty
- Add a multipop(k) method that pops *k* elements (or empties the stack if there are less than *k* elements on the stack)
  - standard operations are $O(1)$
  - multipop is worst case $O(n)$

Aggregate Analysis

- Given a sequence of m stack operations on an initially empty stack
  - Naïve calculation:
    - At most m elements on the stack at one time
    - Therefore, worst case cost is $\sim m \times m = O(m^2)$
  - Better analysis:
    - Combined number of steps of pops and multi-pops is smaller or equal to the number of steps of pushes
    - Therefore: combined number of steps is at most m
    - Therefore: amortized costs $O(1)$

Aggregate Analysis

- Counters implemented as binary array
  - Interested in calculating the bit-flips

0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

  - Single increment can flip all the bits
  - Array with k binary registers has $2^k - 1$ increments
    - Does this mean that number of bit-flips for k increments is $\Theta(k)$ per increment?

Aggregate Analysis

- Write down binary numbers in order
- Observe
  - Least Significant Bit (LSB) flips every time
  - Second LSB flips every other time
  - Third LSB flips every 2^2 times
  - …

```
000000
000001
000010
000011
000100
000101
000110
000111
001000
001001
001010
001011
001100
001101
001110
```

Aggregate Analysis

- Assume *n* increment operations with an arbitrary starting counter value
- Then number of bit-flips is less than
  - $\sum_{i=0}^{k-1} \lfloor \frac{n}{2^i} \rfloor$
    - $< \sum_{i=0}^{k-1} \frac{n}{2^i}$
    - $< n \sum_{i=0}^{\infty} \frac{1}{2^i}$
    - $= n \times 0.11111\ldots_2 = 2n$

Aggregate Analysis

- Aggregate cost of *n* increment operations is therefore
  - $< 2n$
  - $= \Theta(n)$
- Average cost per increment is 2

Accounting Method

- Stack costs:
  - Push: 1
  - Pop: 1
  - Multipop(k): $\min(k, s)$ where s is the number of elements in the stack
- Charges:
  - Push: 2
  - Pop: 0
  - Multipop(k): 0

Accounting Method

- Show that charges will pay for all operations.
  - push pays for itself and for removing the element
  - since we cannot remove elements that have not been pushed, charged amount is always sufficient

Accounting Method

- Counter:
  - Charge 2 for every bit set to one
  - This allows us to set the bit and to reset the bit
- To calculate costs of increment operation:
  - Observe: increment only sets one bit to 1.
- Therefore:
  - n increments cost at most $2n$ bit-flips

```
000000
000001
000010
000011
000100
000101
000110
000111
001000
001001
001010
001011
001100
001101
001110
001111
010000
010001
010010
…
```

Potential Method

- Represent charges as potential energy
  - Potential function Φ maps each state of the data structure to a number
  - Amortized cost of an operation:
    - actual cost + change in potential
  - Implies:
    - Amortized costs of *n* operations
      - actual costs of *n* operations + change in potential

Potential Method

- Stack:
  - Potential = number of elements in stack
    - Potential can never be less than zero
  - Amortized cost of a stack operation
    - Push: cost plus change in potential = 1 + 1
    - Multipop / pop: cost plus change in potential = k - k

Potential Method

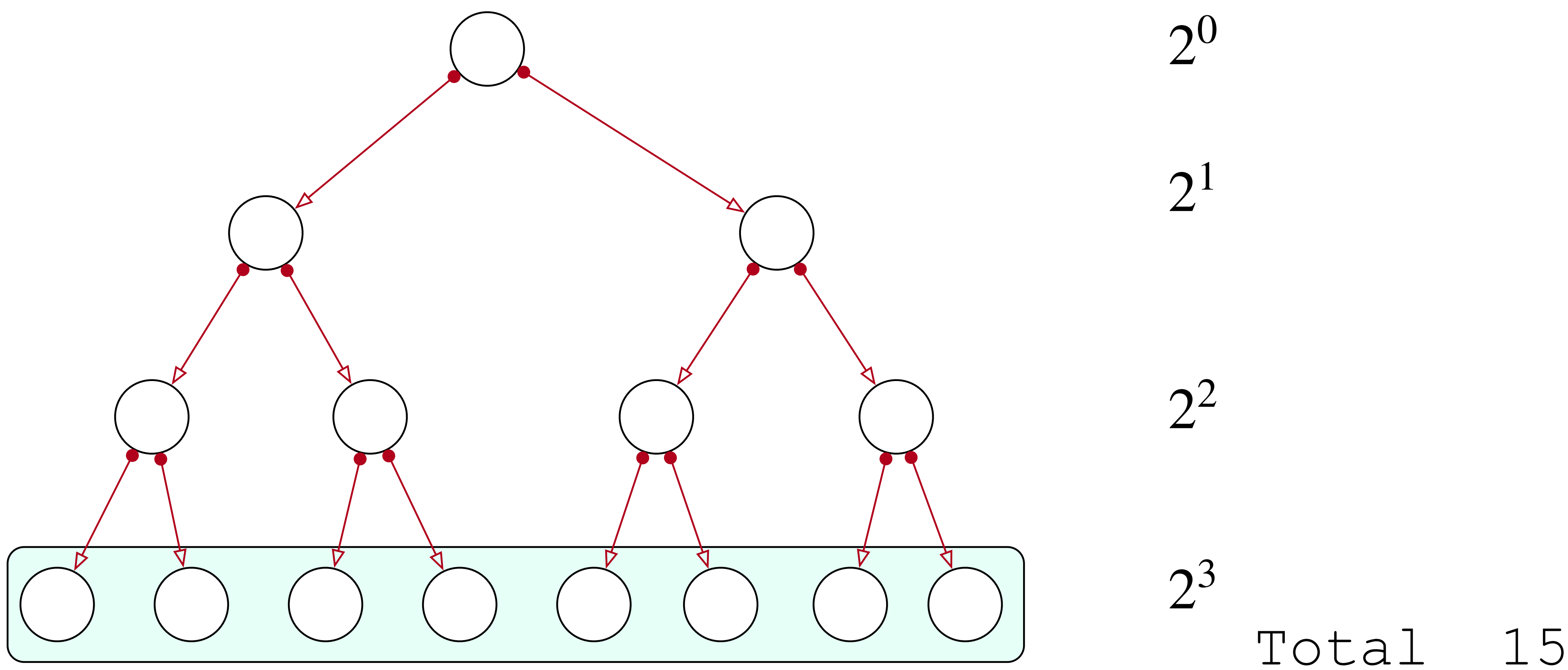
- Counter:
  - Potential = number of bits set (= number of one bits)
  - Amortized cost of an increment:
    - If i^{th} increment resets t_i bits:
      - Cost is $1 + t_i$
      - Amortized cost is
        - cost + potential change
          $\leq (t_i + 1) + (1 - t_i) = 2$

Binary Trees and Heaps

Thomas Schwarz, SJ

Behavior of Trees

- A full binary tree of depth n has
  - $1 + 2 + 2 \cdot 2 + 2 \cdot 2 \cdot 2 + \ldots + 2^{n-1}$
  - $= (111\ldots1)_2 = 2^n - 1$ places



Behavior of Trees

- Reversely:
  - To store *m* elements in a binary tree:
    - Need a tree of depth d such that
      - $2^{(d-1)} - 1 \leq m < 2^d - 1$
    - Equivalent to
      - $2^{d-1} \leq m + 1 < 2^d$
      - $d - 1 \leq \log_2(m + 1) < d$
      - $d - 1 = \lfloor \log_2(m + 1) \rfloor$

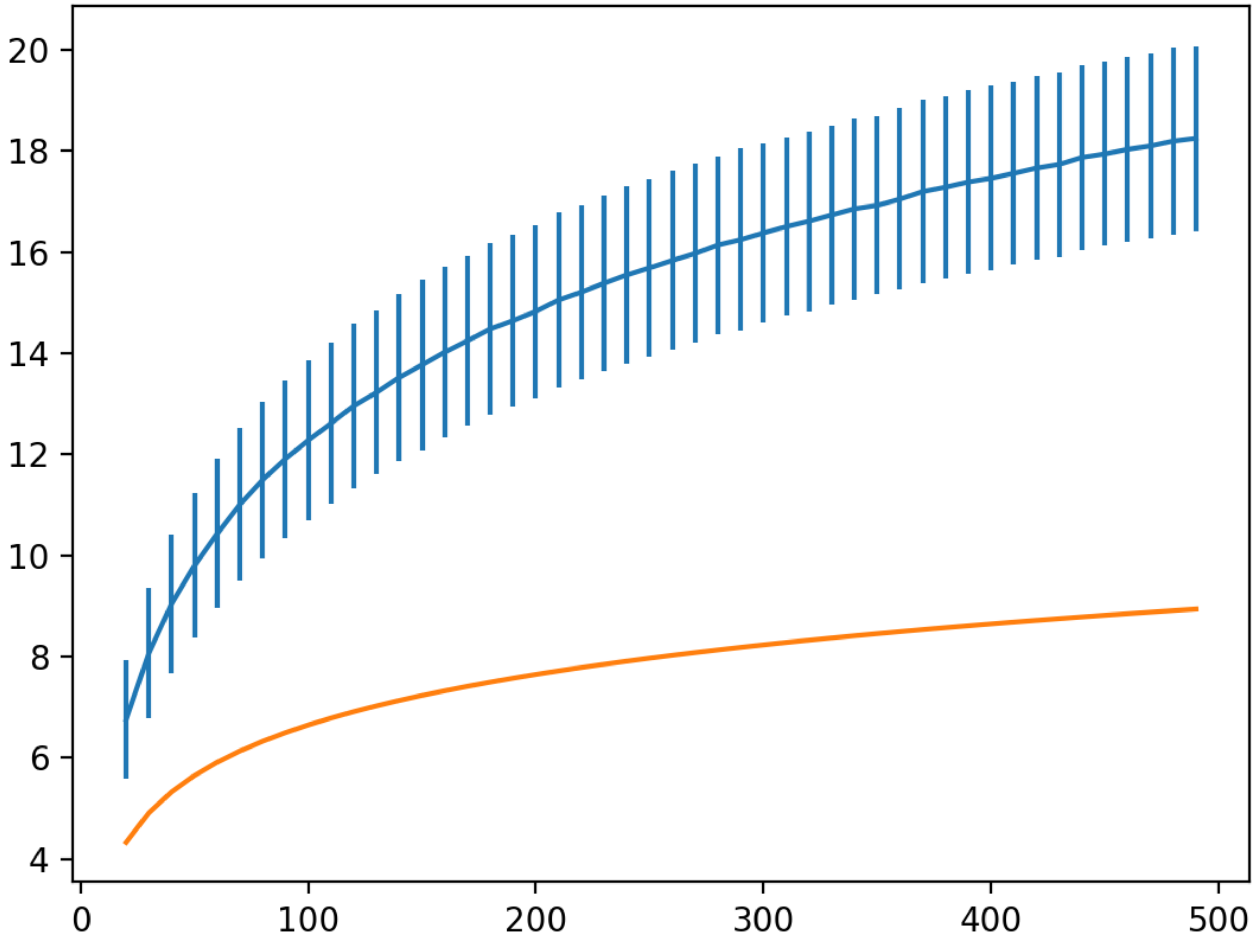
Behavior of Trees

- This parsimony is not natural
  - Random inserts: Trees have much larger depth
  - Self-modifying trees restructure themselves in order to get closer
- Importance:
  - Searching an element takes time ~ to depth
  - Inserting an element takes time ~ to depth

Behavior of Trees

- Experiment:
  - Insert *n* elements into a binary tree
  - Get the depth
  - Repeat 10,000 times
  - Depict mean plus/minus standard deviation

Behavior of Trees



Behavior of Trees

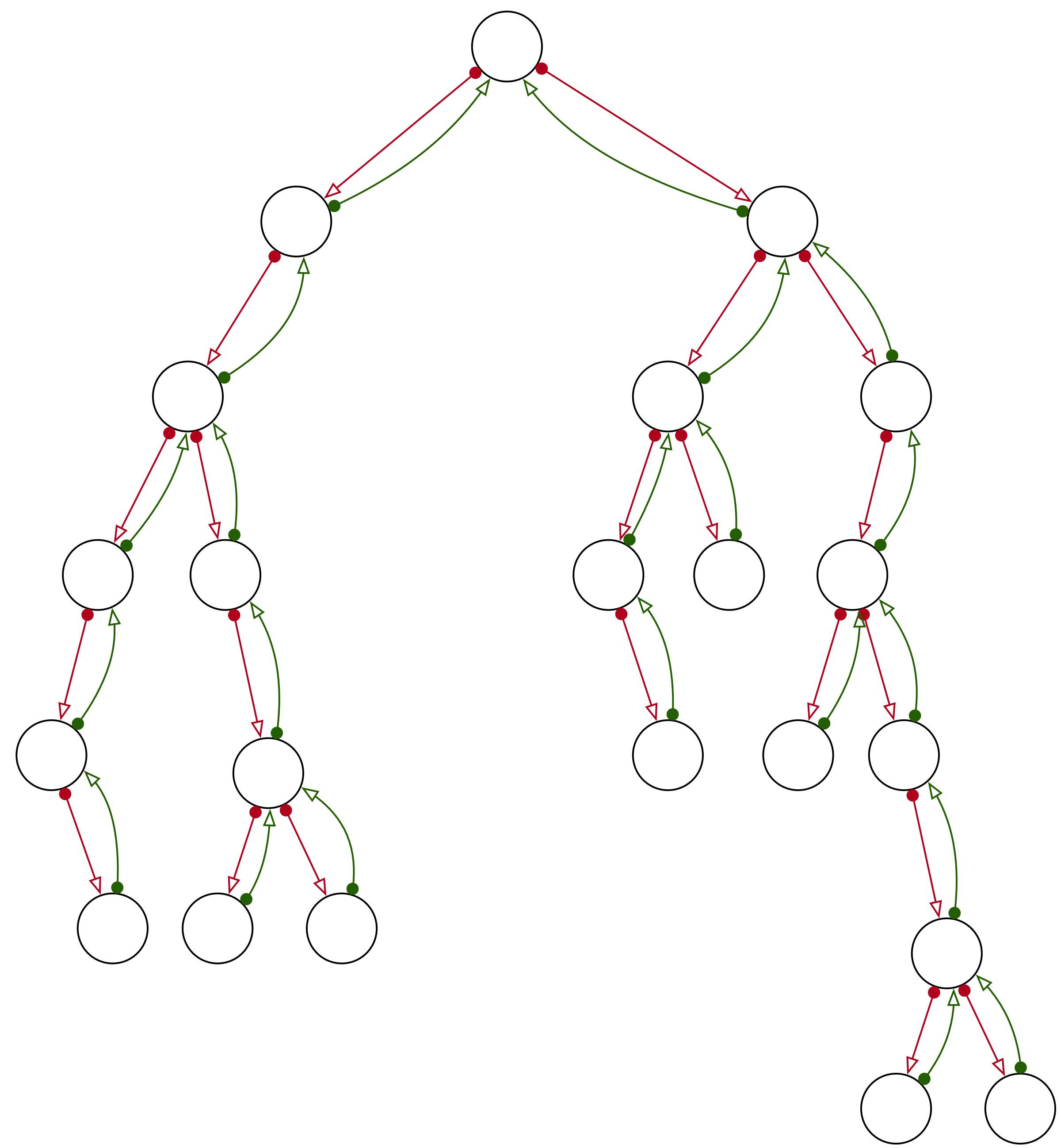
- On average
  - Trees have more than twice necessary depth
- But on average
  - Behavior is still logarithmic
- Theory: Height of a random binary search tree for a random permutation of n elements is
  - $\alpha \log_e(n)$ with $\alpha \approx 4.31107$
  - $= 2.98821 \log_2(n)$
    - Robson 1979 / Devroye 1986

Decorating Binary Search Trees

- General principle for Data Structures:
  - Can store more information in order to improve performance
- Example:
  - Removal of elements from a binary search tree
    - Difficult because we need to find parent
    - Can be made simpler by having a parent pointer

Binary Trees with Parent Link

- Each node stores a link to the parent
- For root, link is None
  - Faster deletes at the cost of more storage per node



Binary Trees with Parent Link

- Expand to a key-value store by adding a field for record
- Add a parent link

```
class Node:
    def __init__(self, value, record):
        self.value = value
        self.record = record
        self.up, self.left, self.right = None, None, None

    def __repr__(self):
        return "Node : {}, Value: {}, Record: {},
                 Left: {}, Right: {}, Up: {}".format(
                    hex(id(self)), self.value, self.record,
                    hex(id(self.left)), hex(id(self.right)),
                    hex(id(self.up)))
```

Binary Trees with Parent Link

- We have to maintain the up link:

```
def insert(self, value, record):
        new_node = Node(value, record)
        if not self.root:
            self.root = new_node
        else:
            current = self.root
            while True:
                if value < current.value:
                    if current.left:
                        current = current.left
                    else:
                        current.left = new_node
                        new_node.up = current
                        return
```

Binary Trees with Parent Link

- But deleting a record is still not trivial
  - Special case when
    - the tree is empty

```
def remove(self, value):
    if not self.root:
        return False
```

Binary Trees with Parent Link

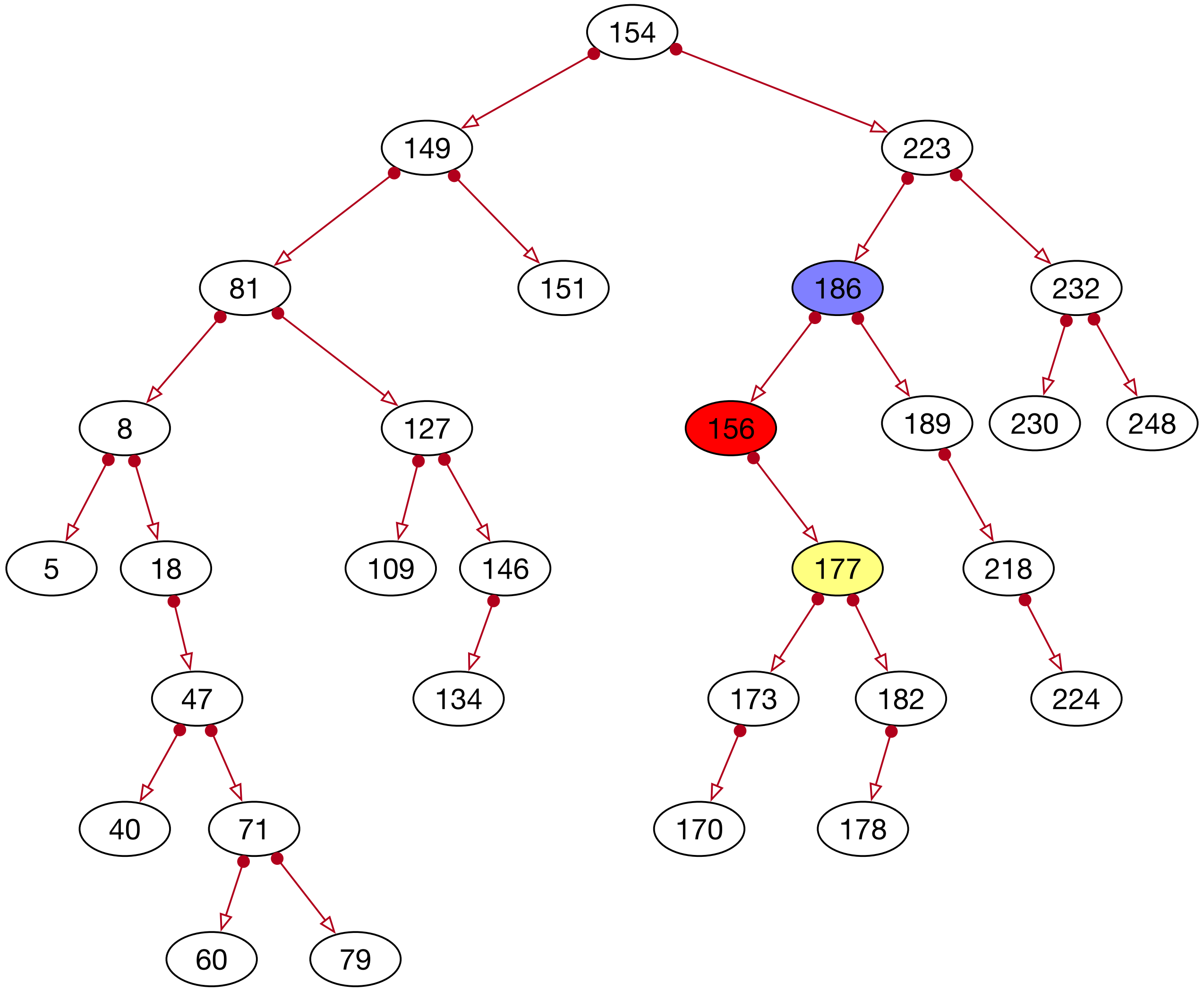
```
def remove(self, value):
        if not self.root:
            return False
        current = self.root
        while True:
            if not current:
                return False
            if value == current.value:
                break
            if value < current.value:
                current = current.left
            else:
                current = current.right
            if current == None:
                return False
        to_delete = current
```

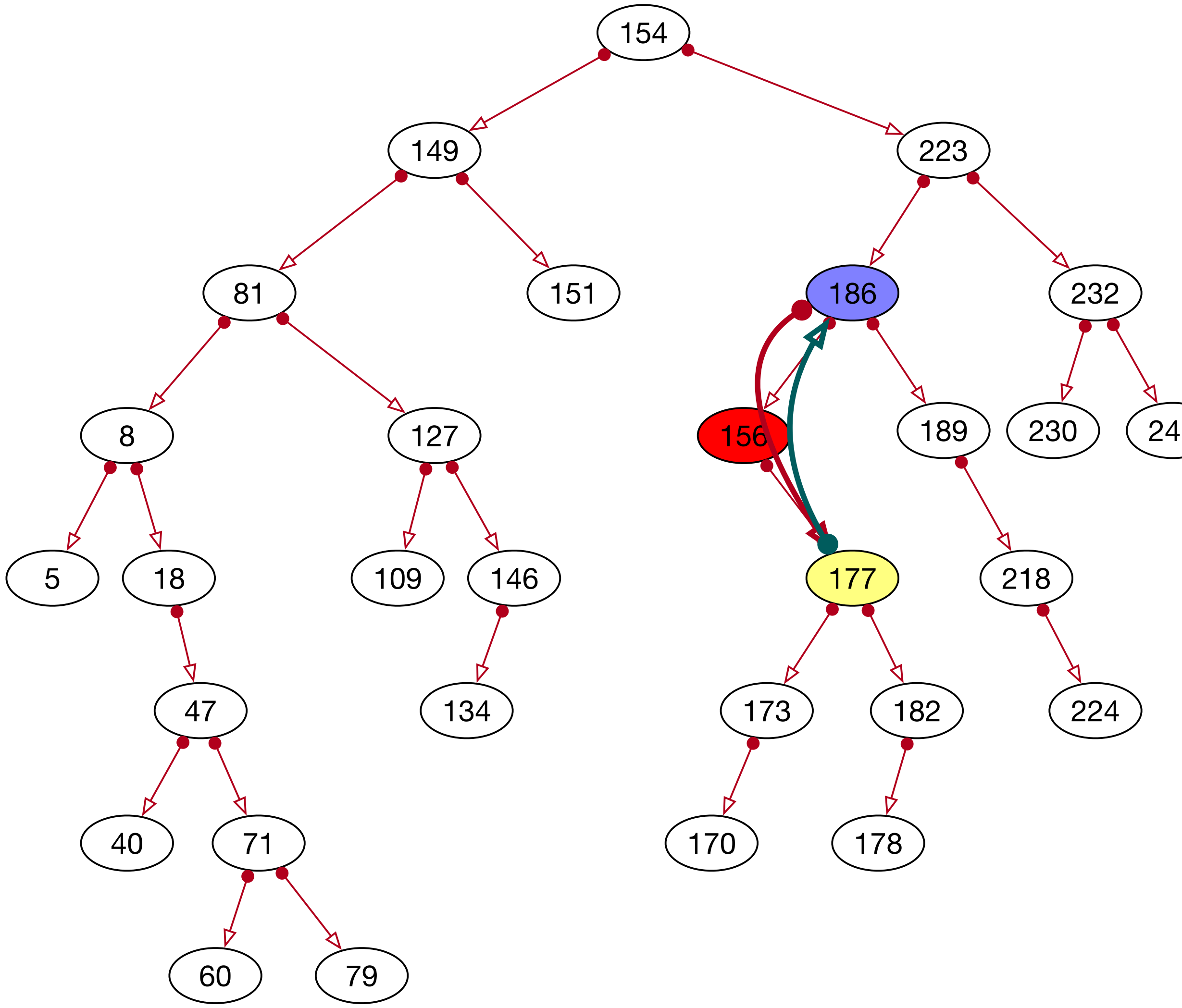
Binary Trees with Parent Link

- We still need to make additional case distinctions
  - But we no longer need a stack to keep track of the nodes
  - Case distinctions:
    - No children:
      - Just delete (unless we are deleting the root)
    - One child
    - Two children

Binary Trees with Parent Link

- Removing node with one child
- Move child up and reset **two** links





Binary Trees with Parent Link

- Special case if parent is root

```
elif not to_delete.left and to_delete.right:
            # node has only a right child
            parent = to_delete.up
            if not parent:
                self.root = to_delete.right
                return True
            else:
                if parent.left == to_delete:
                    parent.left = to_delete.right
                    to_delete.right.up = parent
                else:
                    parent.right = to_delete.right
                    to_delete.right.up = parent
                return True
```

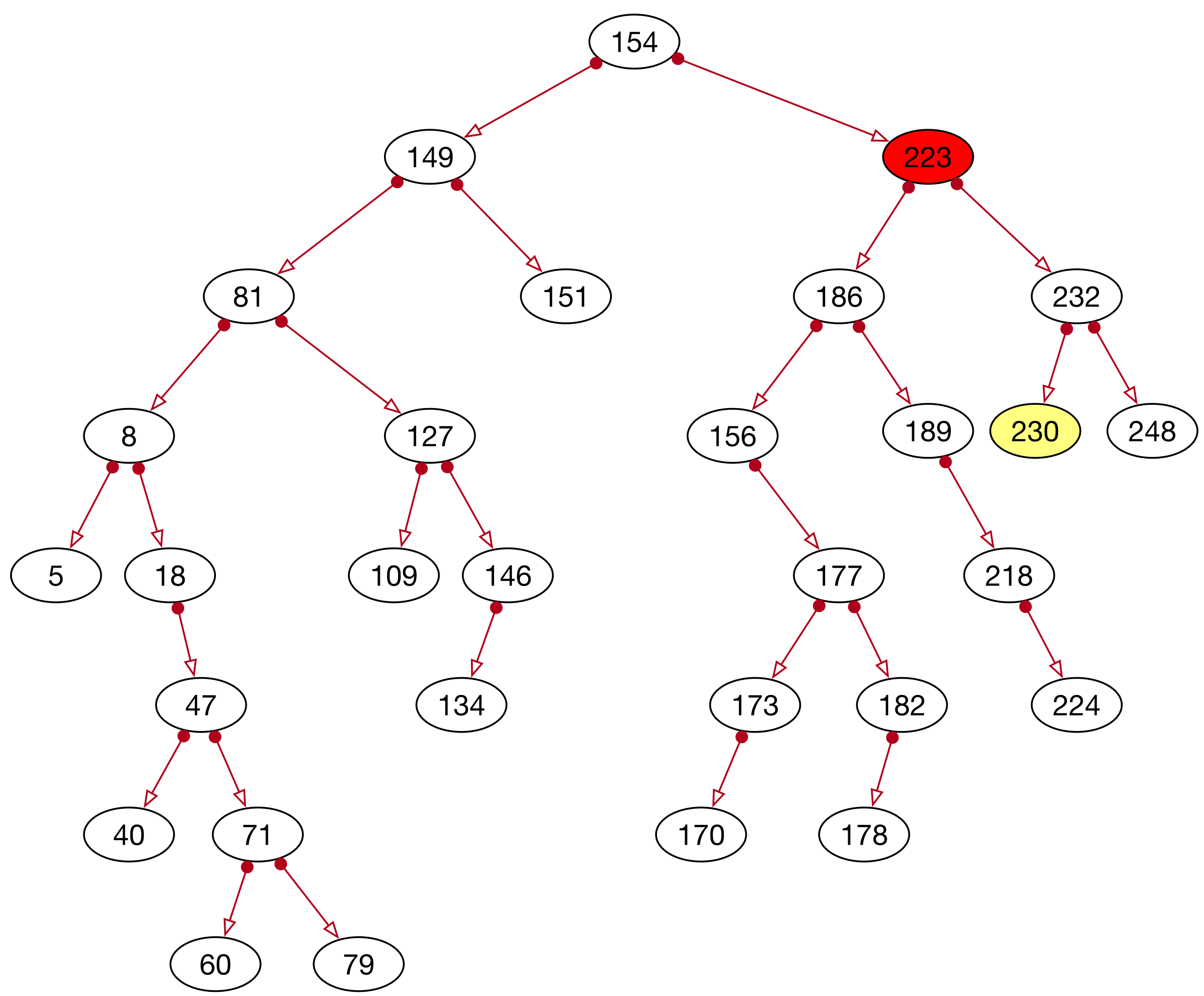
Binary Trees with Parent Link

- Otherwise: reset two links

```
elif not to_delete.left and to_delete.right:
            # node has only a right child
            parent = to_delete.up
            if not parent:
                self.root = to_delete.right
                return True
            else:
                if parent.left == to_delete:
                    parent.left = to_delete.right
                    to_delete.right.up = parent
                else:
                    parent.right = to_delete
                    to_delete.right.up = parent
                return True
```

Binary Trees with Parent Link

- Two children:
    - Identify the next node in-order traversal



Binary Trees with Parent Link

- Two children:
  - Find the next node in in-order traversal:
    - Go to the right: `current.right`
    - Then go always to the left

```
def min_value_node(a_node):
    current = a_node
    while current.left:
        current = current.left
    return current
```

Binary Trees with Parent Link

- Two nodes

```
elif to_delete.left and to_delete.right:
            #node has two children
            leaf = Binary_Tree.min_value_node(
                                         to_delete.right)
            save_value = leaf.value
            save_record = leaf.record
            self.remove(leaf.value)
            to_delete.value = save_value
            to_delete.record = save_record
```

Binary Trees with Parent Link

- Safe the values of the resulting leaf

```
elif to_delete.left and to_delete.right:
            #node has two children
            leaf =
             Binary_Tree.min_value_node(to_delete.right)
            print('leaf',leaf)
            save_value = leaf.value
            save_record = leaf.record
            self.remove(leaf.value)
            to_delete.value = save_value
            to_delete.record = save_record
```

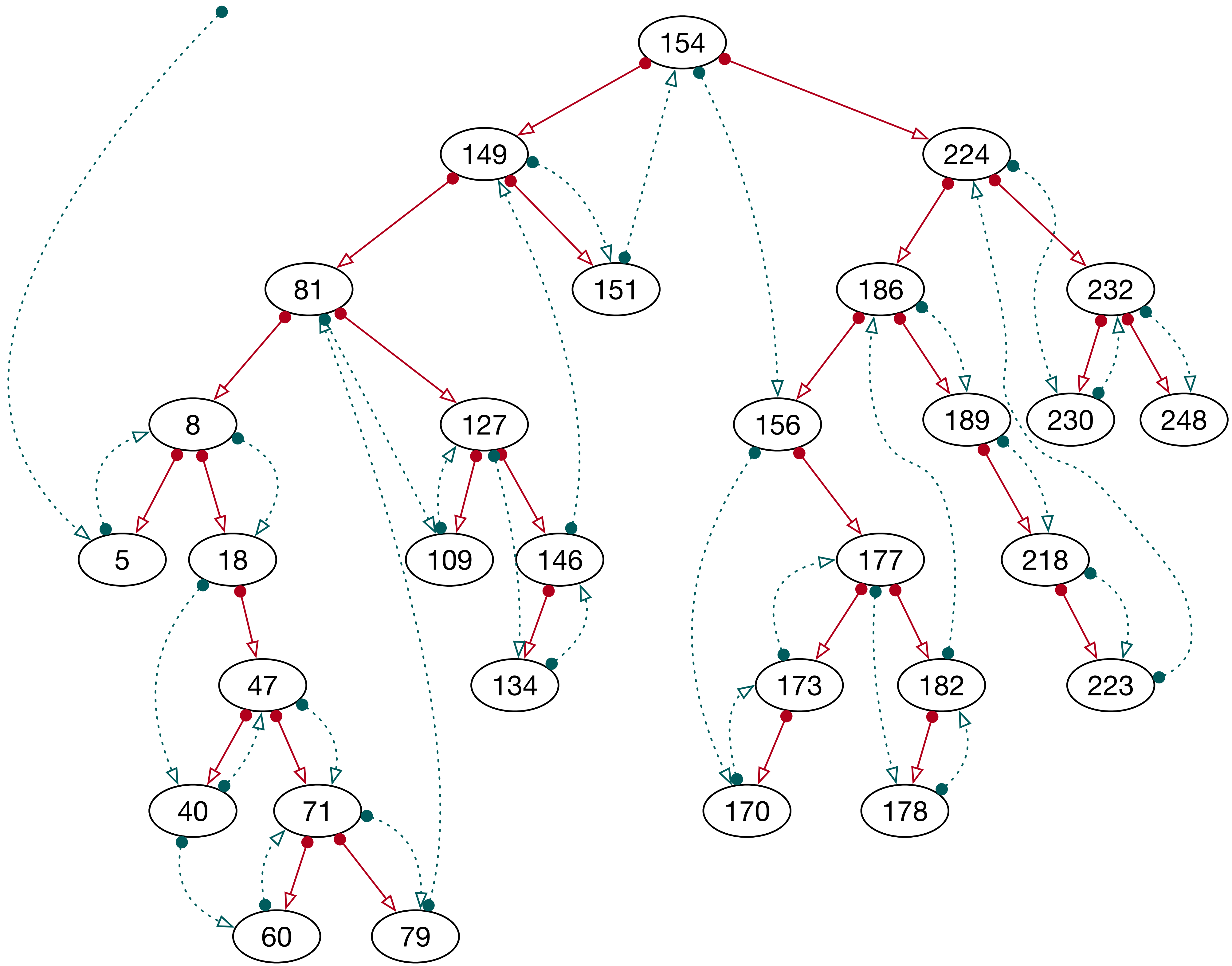
Binary Trees with Parent Link

- Then delete the leaf
  - I cheat by using recursion

```
elif to_delete.left and to_delete.right:
            #node has two children
            leaf =
             Binary_Tree.min_value_node(to_delete.right)
            print('leaf',leaf)
            save_value = leaf.value
            save_record = leaf.record
            self.remove(leaf.value)
            to_delete.value = save_value
            to_delete.record = save_record
```

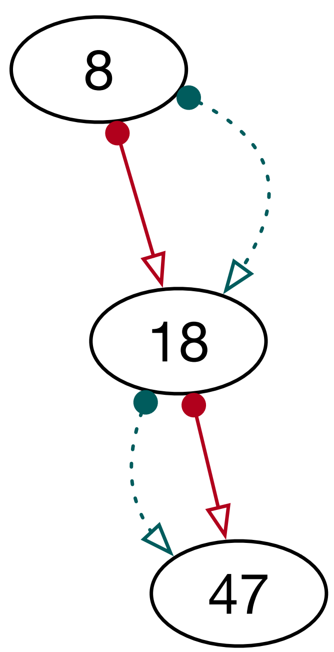
Binary Trees with Parent Link

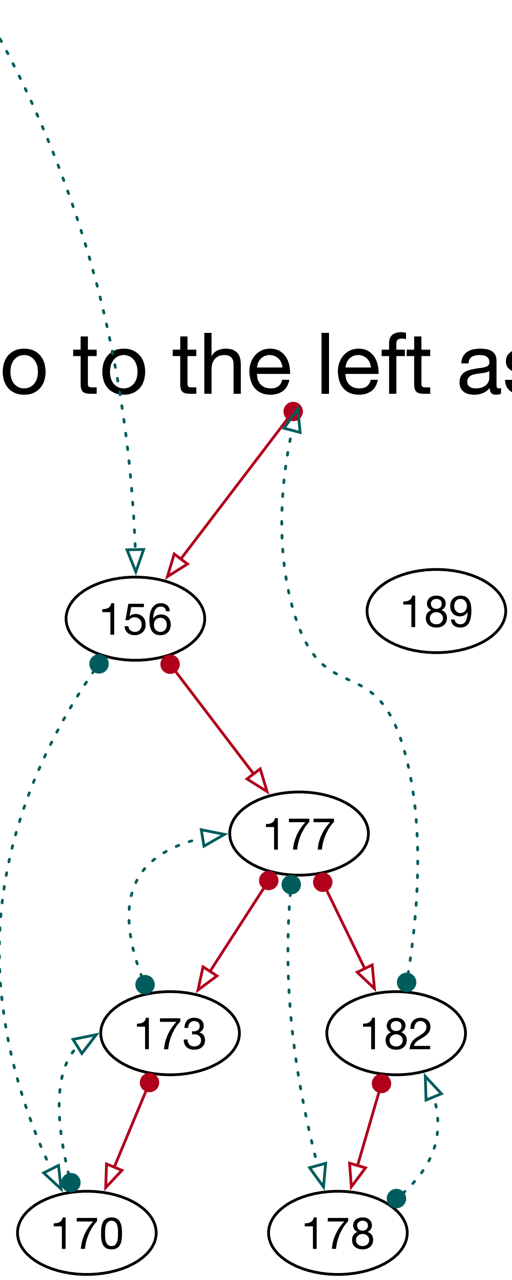
- Non-recursive in-order traversal
  - Here is a tree with an additional set of links for in-order traversal


154
149
224
81
151
186
232
8
127
156
189
230
248
5
18
109
146
177
218
47
134
173
182
223
40
71
170
178
60
79

Binary Trees with Parent Link

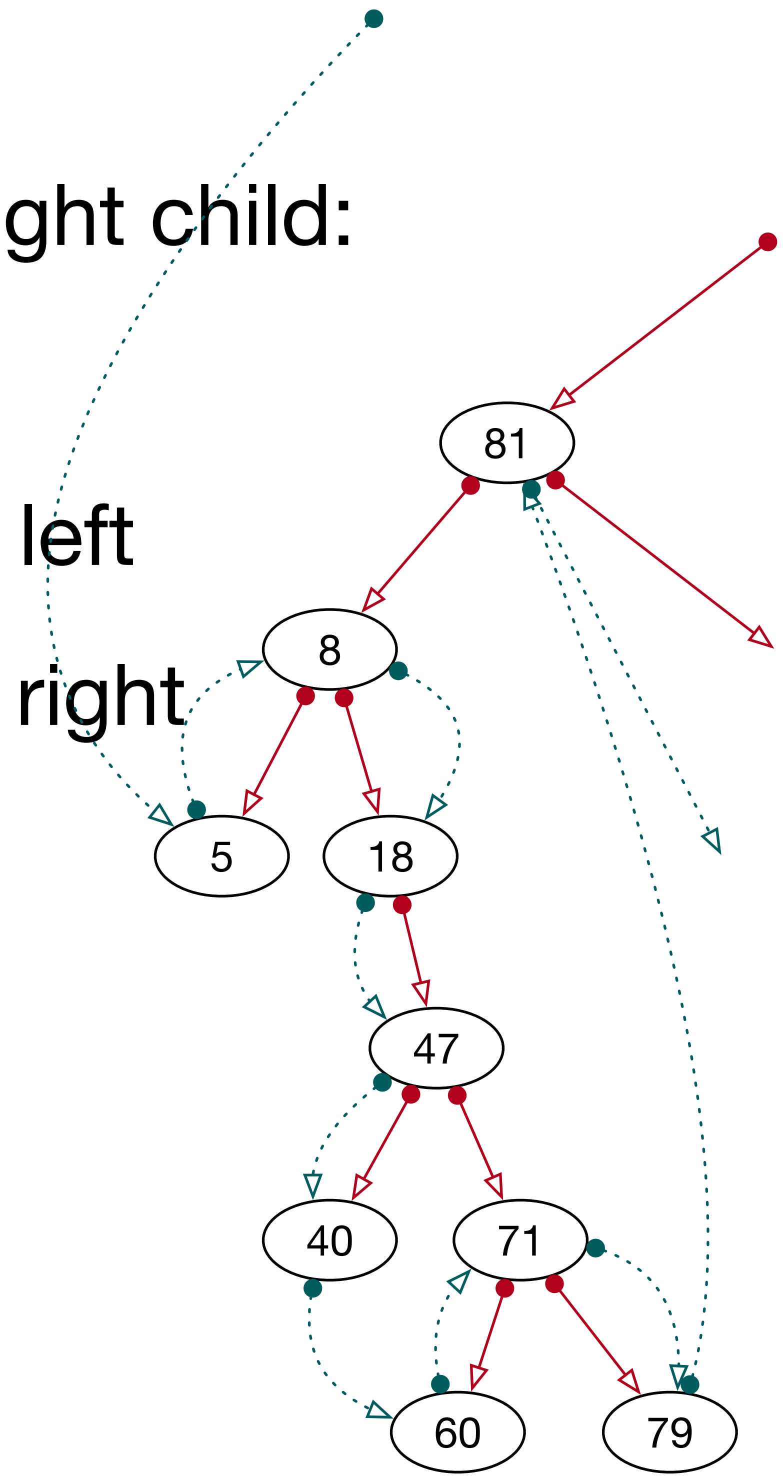
- What is the next node:
  - If the node has a right child:
    - Go one to the right, then go to the left as much as possible





Binary Trees with Parent Link

- What is the next node if there is no right child:
    - If parent is to the left:
        - Follow parents if they are to the left
        - Then take the first parent to the right



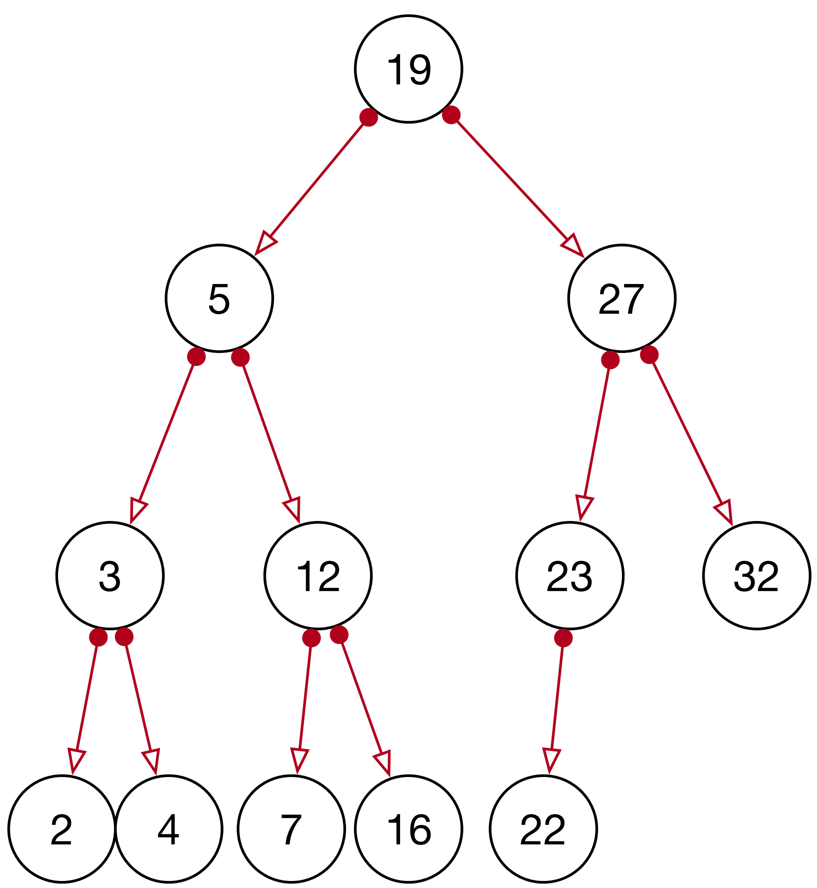
Binary Trees with Parent Link

- Thus:
  - Can do in-order traversal without a stack or recursion

Binary Trees using Arrays

Using Arrays

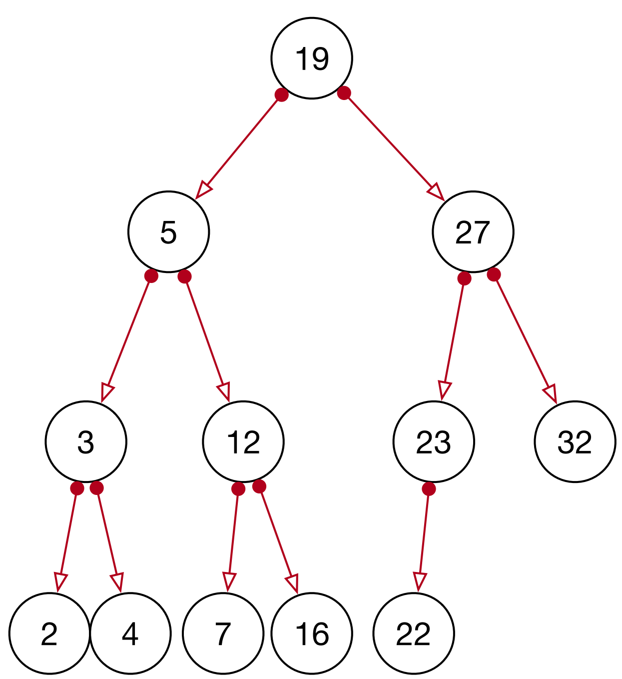
- In a tree, each node has up to two children
  - Can organize nodes in an array
    - Leave first spot open



—	19	5	27	3	12	23	32	2	4	7	16	22
	1	2	3	4	5	6	7	8	9	10	11	12

Using Arrays

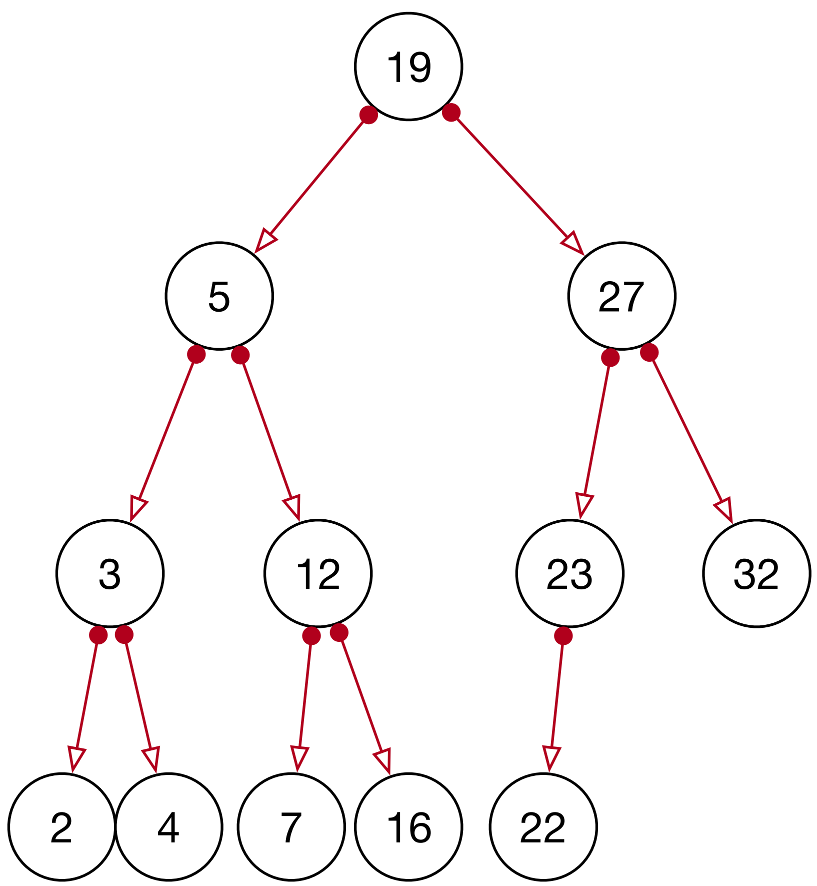
- Left child of node at index i
    - Located at index $2i$
- Right child of node at index i
    - Located at index $2i + 1$



–	19	5	27	3	12	23	32	2	4	7	16	22
	1	2	3	4	5	6	7	8	9	10	11	12

Using Arrays

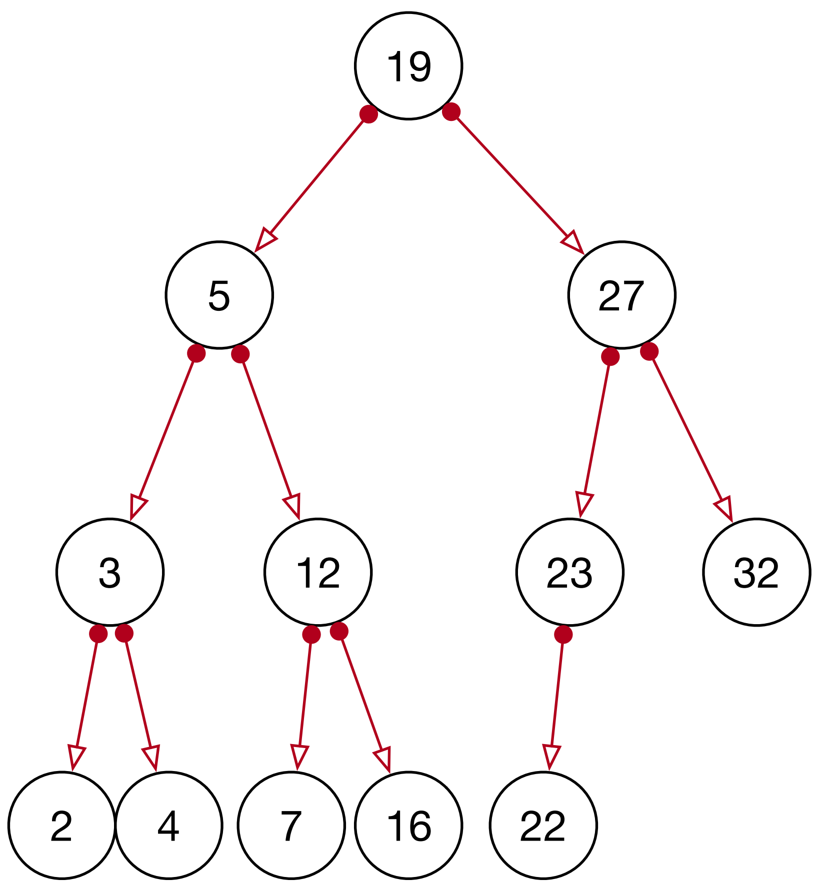
- Parent of node at index i is located at index $i//2$
  - Mathematical notation: $\lfloor \frac{i}{2} \rfloor$



—	19	5	27	3	12	23	32	2	4	7	16	22
	1	2	3	4	5	6	7	8	9	10	11	12

Using Arrays

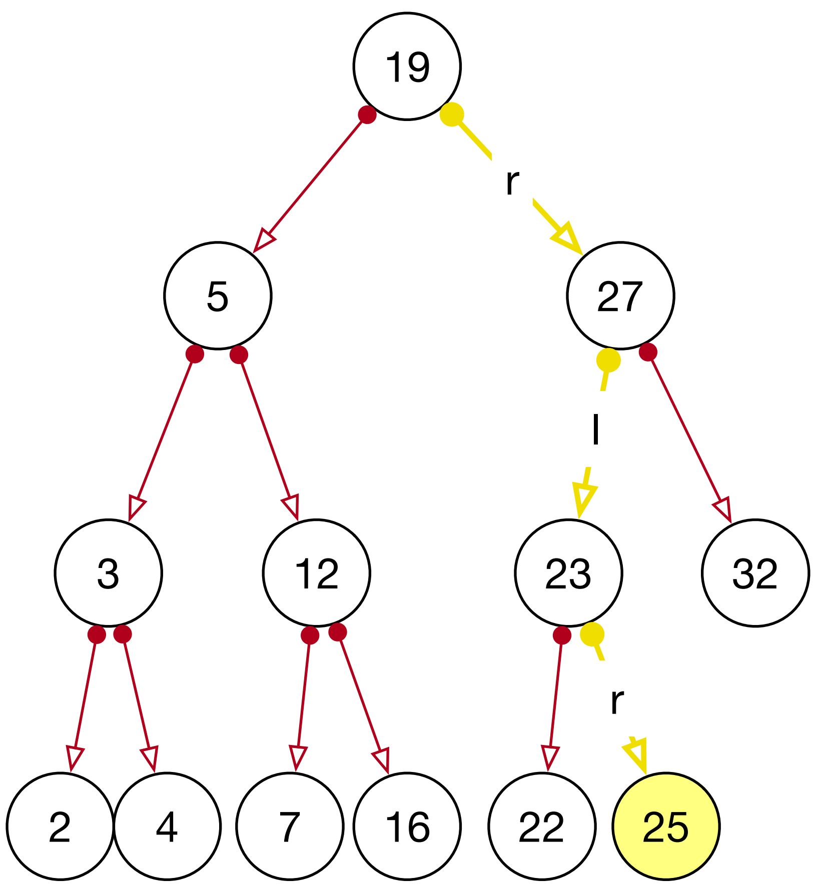
- Right children are at odd indices, left children are even indices



—	19	5	27	3	12	23	32	2	4	7	16	22
	1	2	3	4	5	6	7	8	9	10	11	12

Using Arrays

- We can calculate the index if we are given a sequence of directions

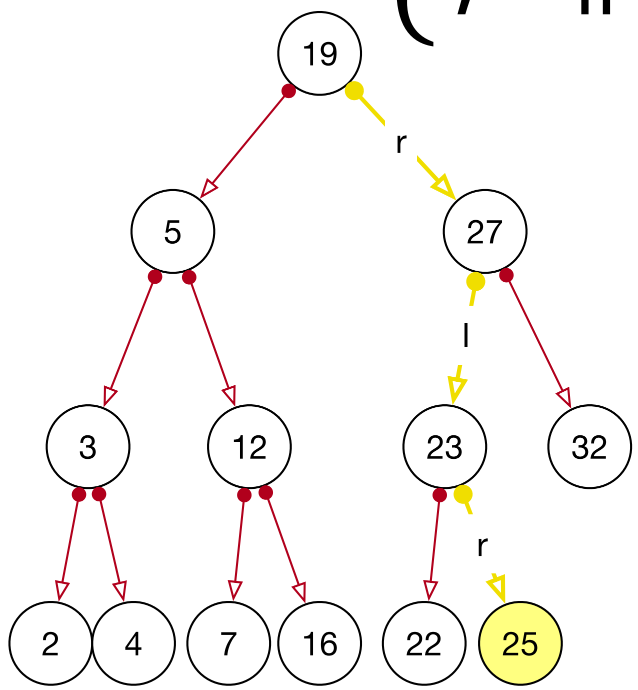


—	19	5	27	3	12	23	32	2	4	7	16	22	25
	1	2	3	4	5	6	7	8	9	10	11	12	13

rlr ((1*2+1)*2)*2+1 =13

Using Arrays

- Define $r(n) := 2n + 1, l(n) := 2n$
- Then node is at index $(o_m \circ o_{m-1} \circ \ldots \circ o_2 \circ o_1)(1)$
- where $o_i = \begin{cases} l & \text{if we go left in step } i \\ r & \text{if we go right in step } i \end{cases}$



—	19	5	27	3	12	23	32	2	4	7	16	22	25
	1	2	3	4	5	6	7	8	9	10	11	12	13

rlr ((1*2+1)*2)*2+1 =13

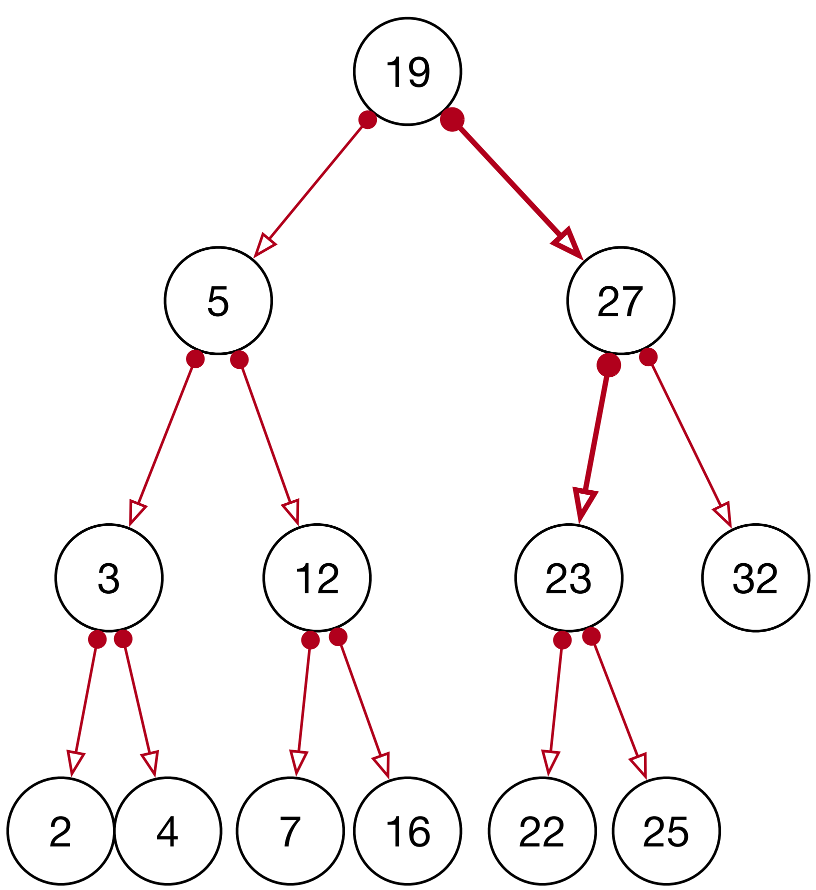
$\boxed{r \circ l \circ r(1)}$

Using Arrays

- Can we do something about the unused first element in the array?
    - We just need to adjust the index: by adding 1 and subtracting 1

Using Arrays

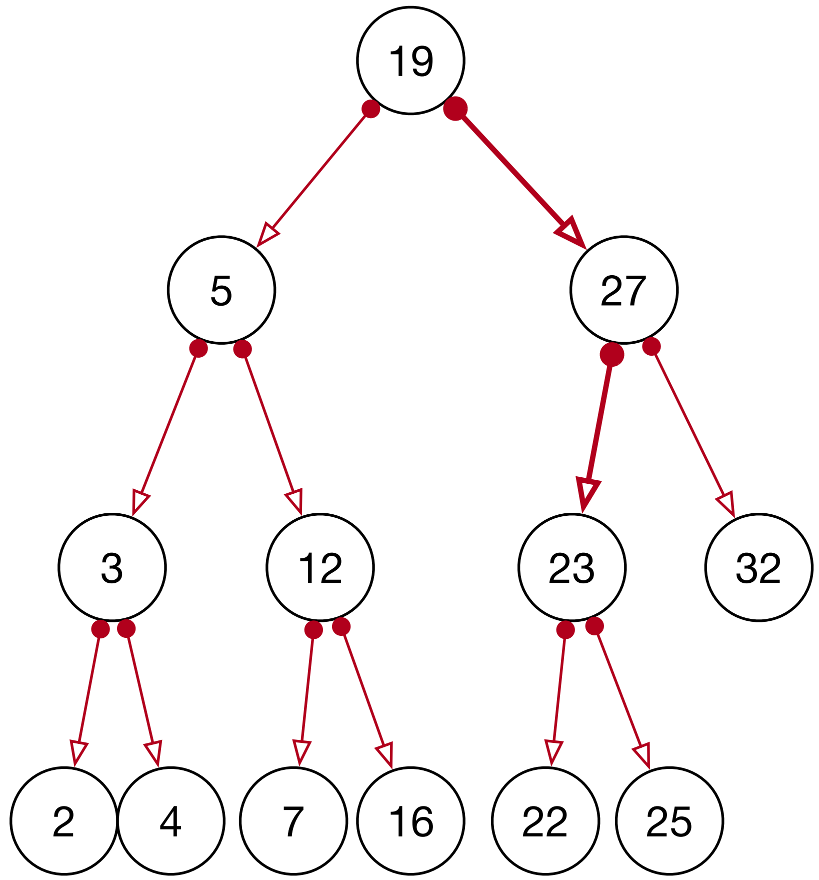
- Children of node i are now $2 \cdot (i+1) - 1 = 2 \cdot i + 1$ and $(2 \cdot (i+1) + 1) - 1 = 2 \cdot i + 2$



19	5	27	3	12	23	32	2	4	7	16	22	25
0	1	2	3	4	5	6	7	8	9	10	11	12

Using Arrays

- Parent of a node located at index i is located
    - at index $\lfloor \frac{i+1}{2} \rfloor - 1$



19	5	27	3	12	23	32	2	4	7	16	22	25
0	1	2	3	4	5	6	7	8	9	10	11	12

Using Arrays

- One advantage:
  - We automatically have a way to find the parent

Priority Queue

- ADS with
  - Insertion
  - Popping maximum element
- Example: insert 5, insert 4, insert 10, pop, insert 7, insert 3, pop, insert 2, pop, pop
  - Returns on insert 5, insert 4, insert 10, **pop**, insert 7, insert 3, pop, insert 2, pop, pop: 10
  - Returns on insert 5, insert 4, insert 10, pop, insert 7, insert 3, **pop**, insert 2, pop, pop: 7
  - Returns on insert 5, insert 4, insert 10, pop, insert 7, insert 3, pop, insert 2, **pop**, pop: 5
  - Returns on insert 5, insert 4, insert 10, pop, insert 7, insert 3, pop, insert 2, pop, **pop**: 4

Priority Queues

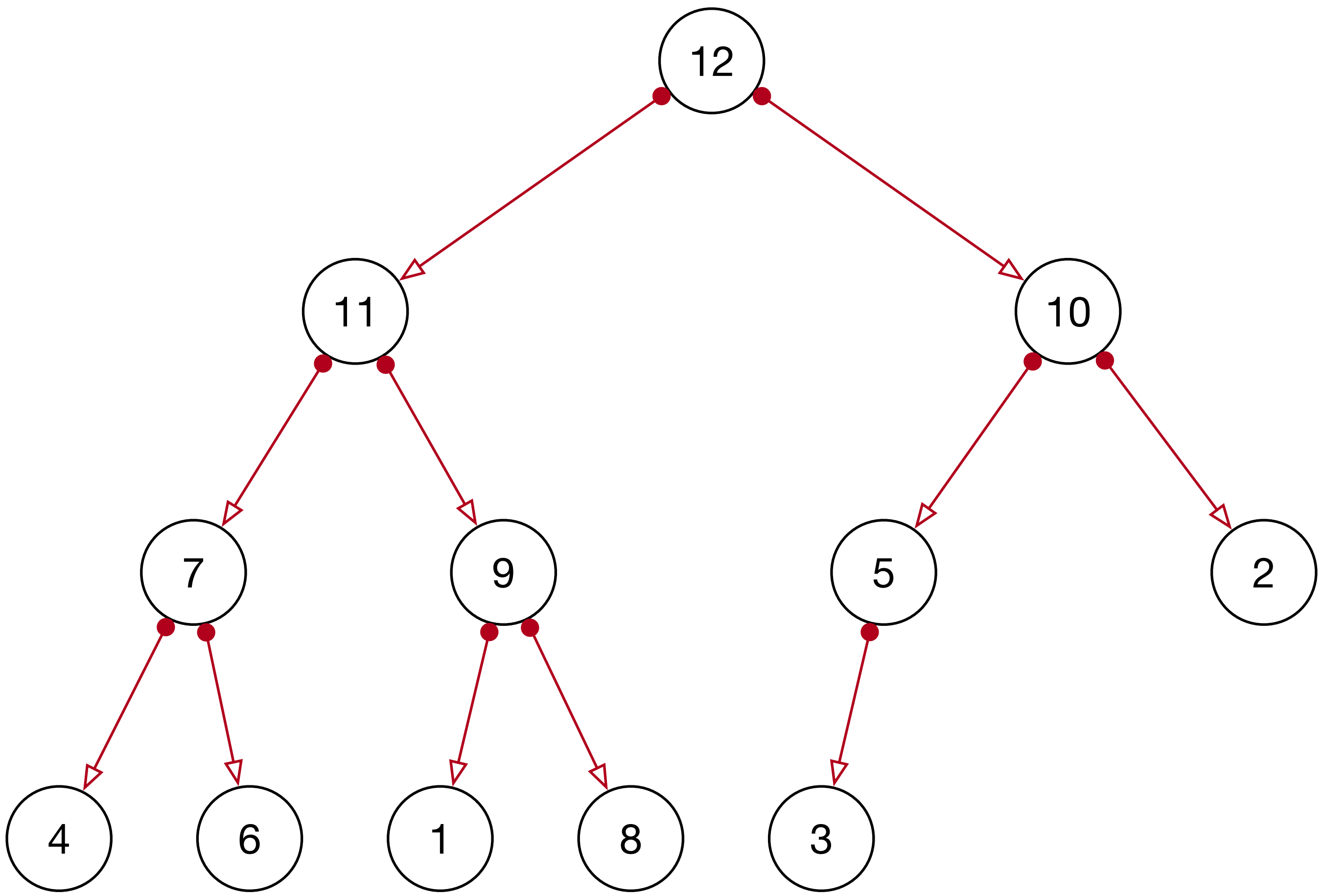
- Simplistic implementation
  - A list
    - Whenever we look for an element, we look for the maximum of the list
    - Run time: Proportional to the length of the list

Priority Queues

- Favorite implementation:
  - Heap:
    - A ***complete*** binary tree
      - Tree is maximum balanced
    - That is **partially** ordered

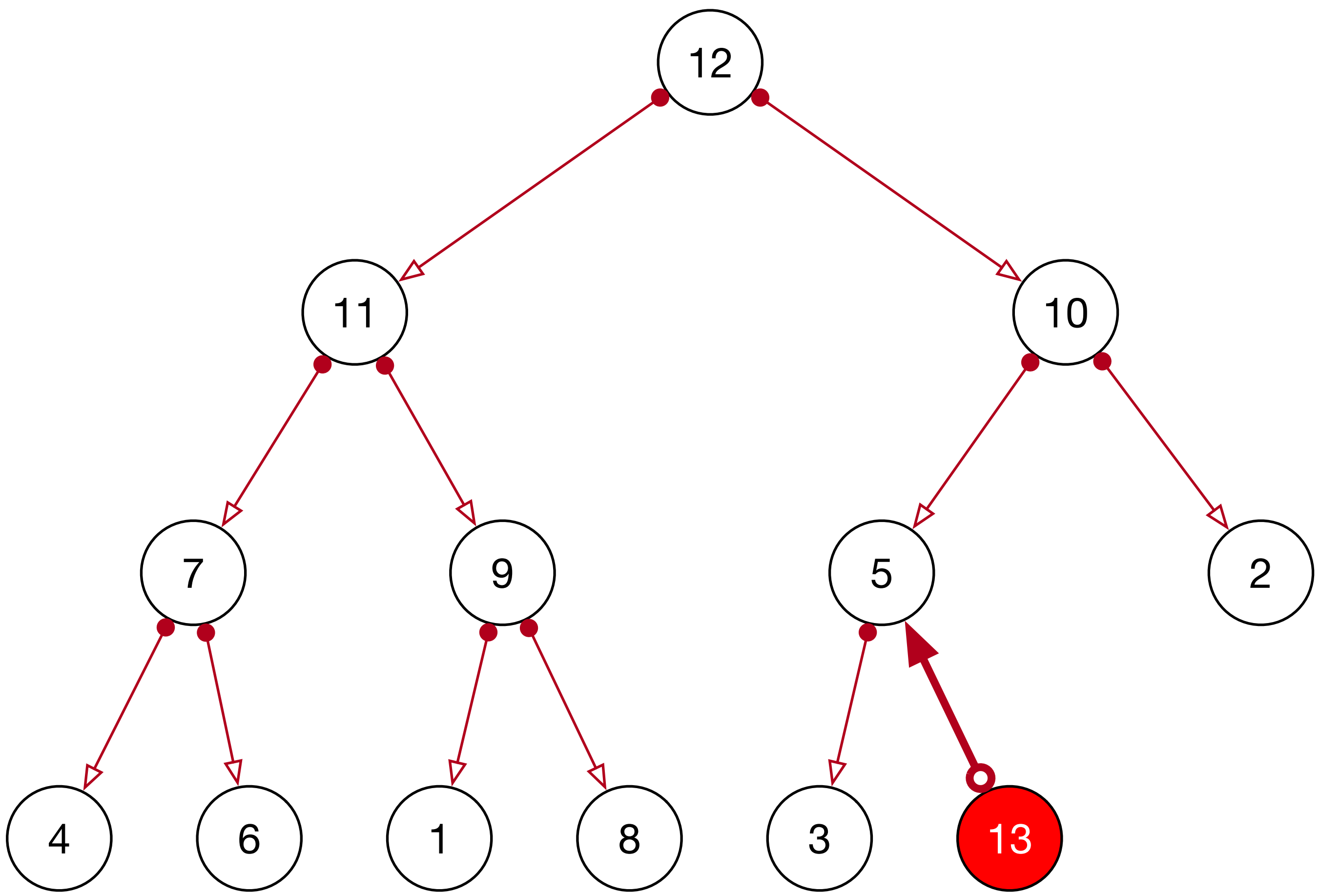
Priority Queues

- Heaps as binary tree
  - Complete:
    - No nodes missing
    - Last generation filled from left
  - Partially ordered:
    - parent has larger value than child



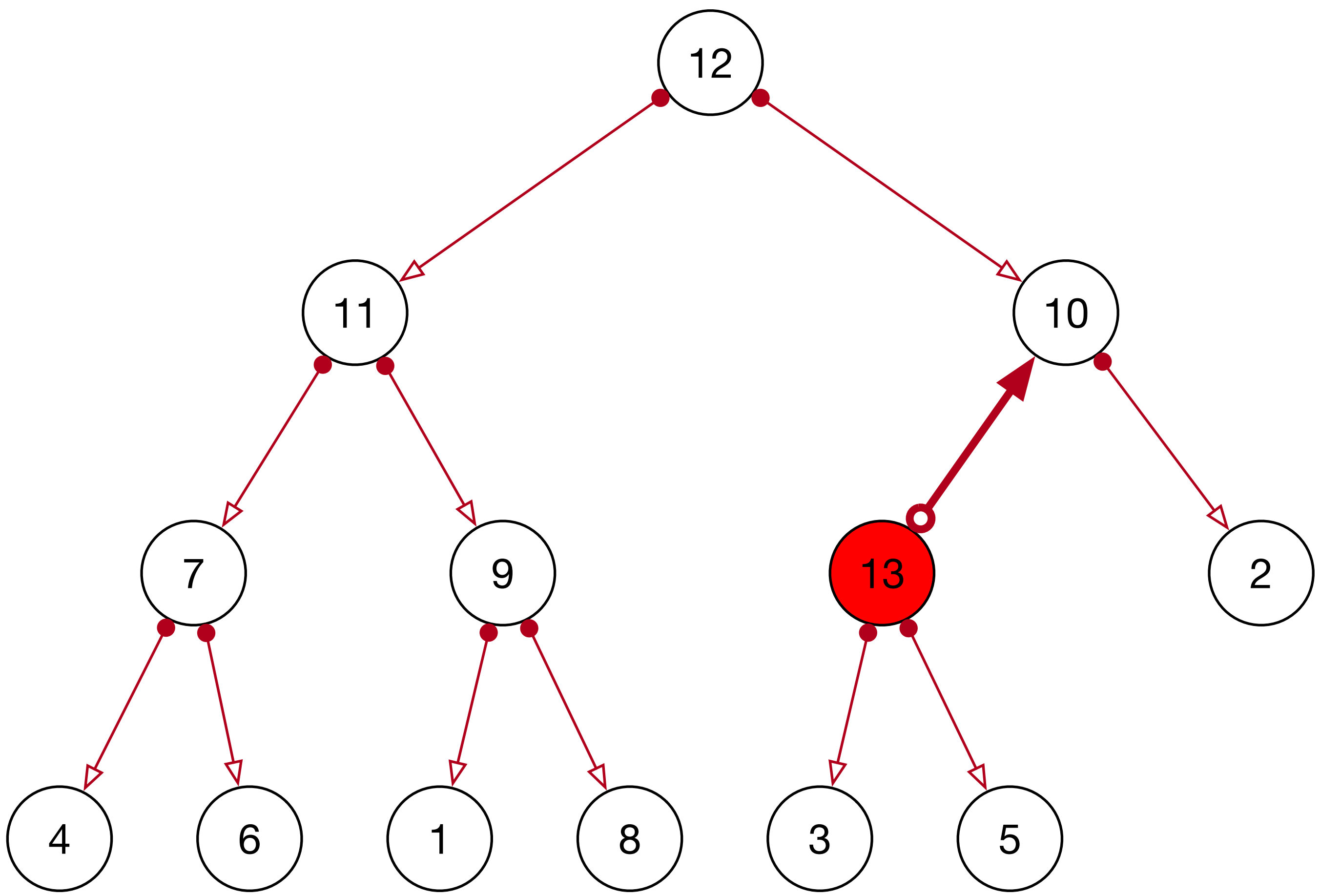
Priority Queues

- Operations: Insertion
  - Insert at the next spot
  - If the new node is larger than the parent:
    - swap with parent



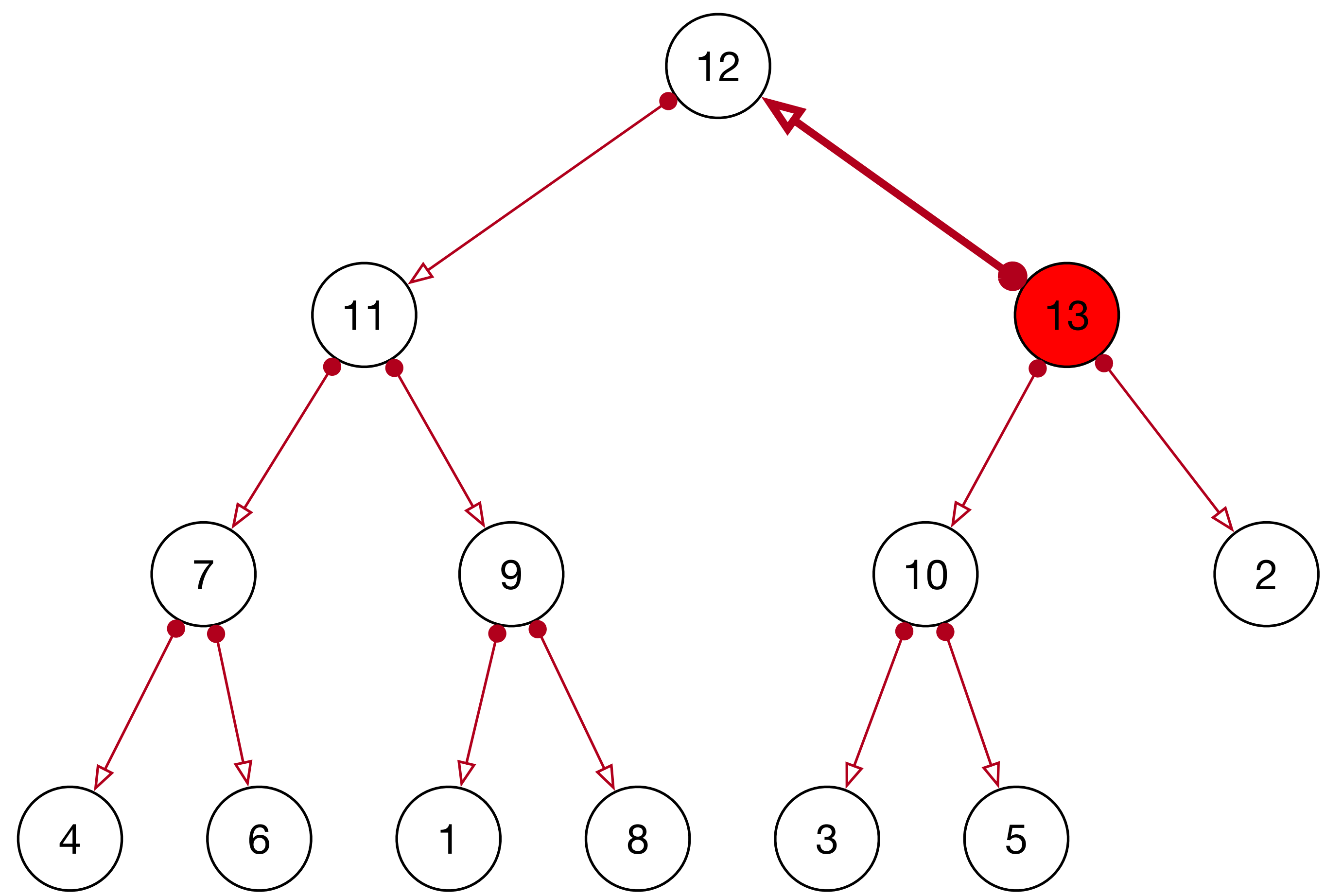
Priority Queues

- This is repeated
  - if necessary



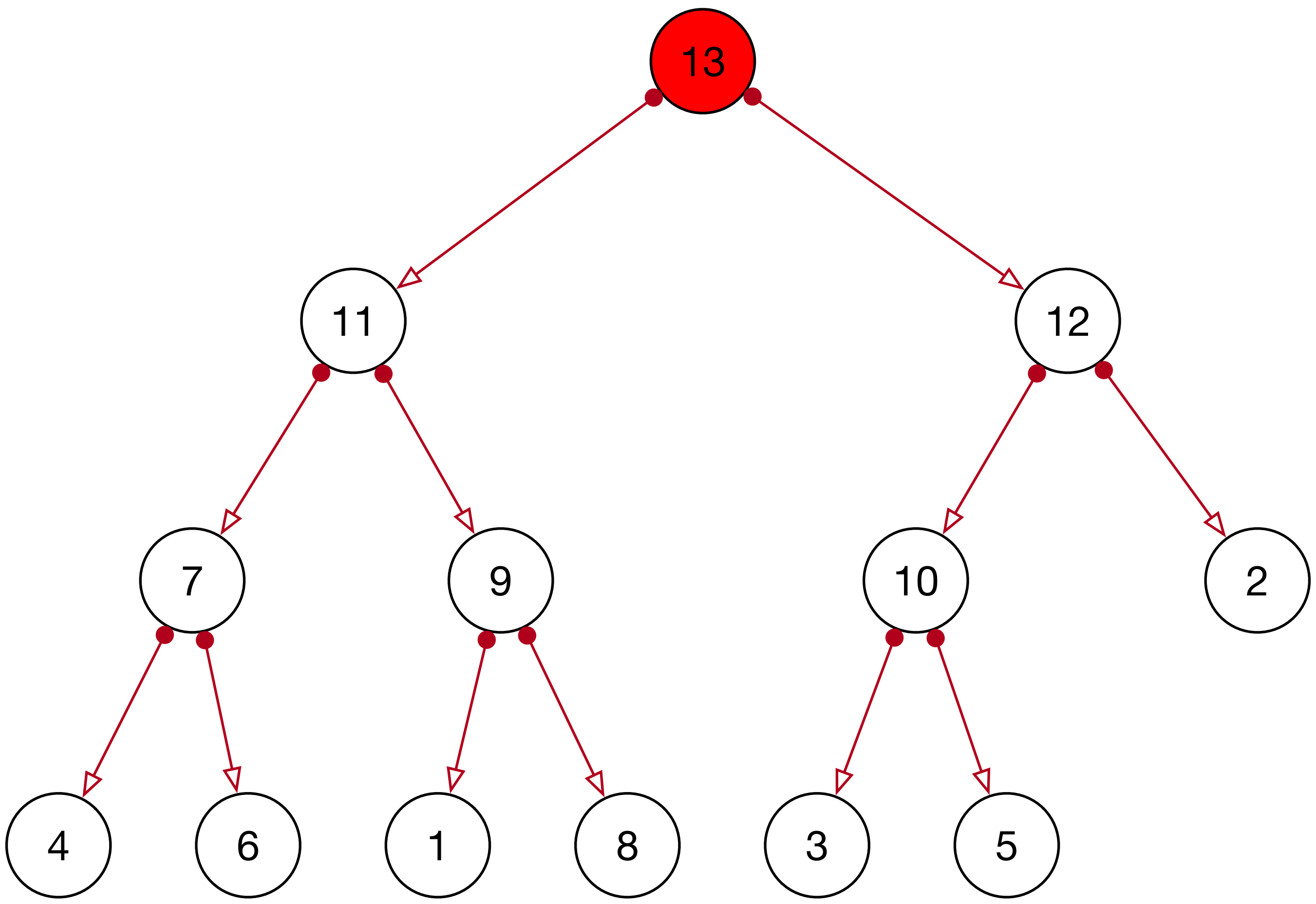
Priority Queues

- Notice:
  - The only violation of order can be with parent



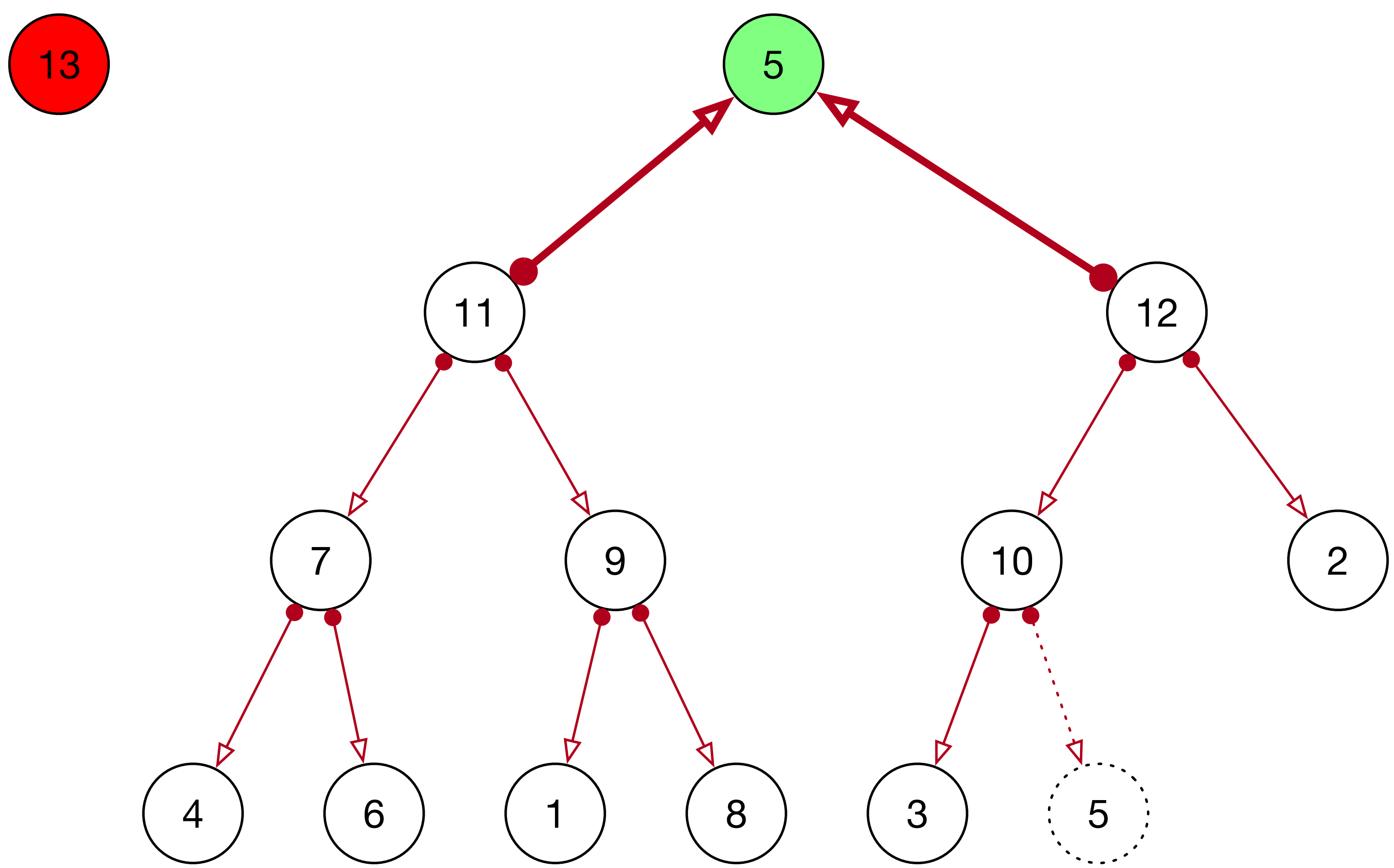
Priority Queues

- There are at most $\log_2(n)$ swaps
  - Compared to n



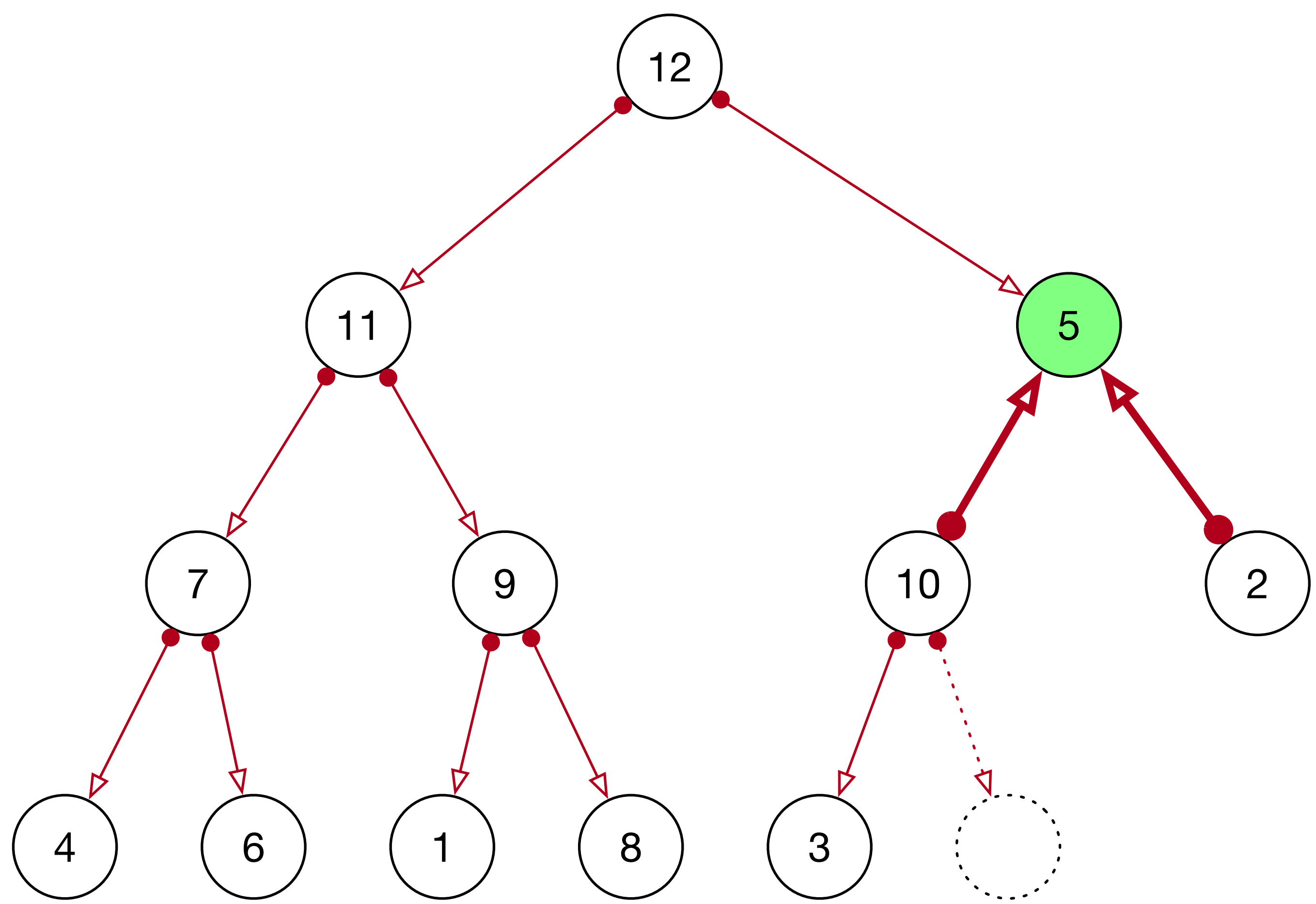
Priority Queues

- Remove Maximum:
  - Maximum is at the top, remove it
  - Move last element into the top position



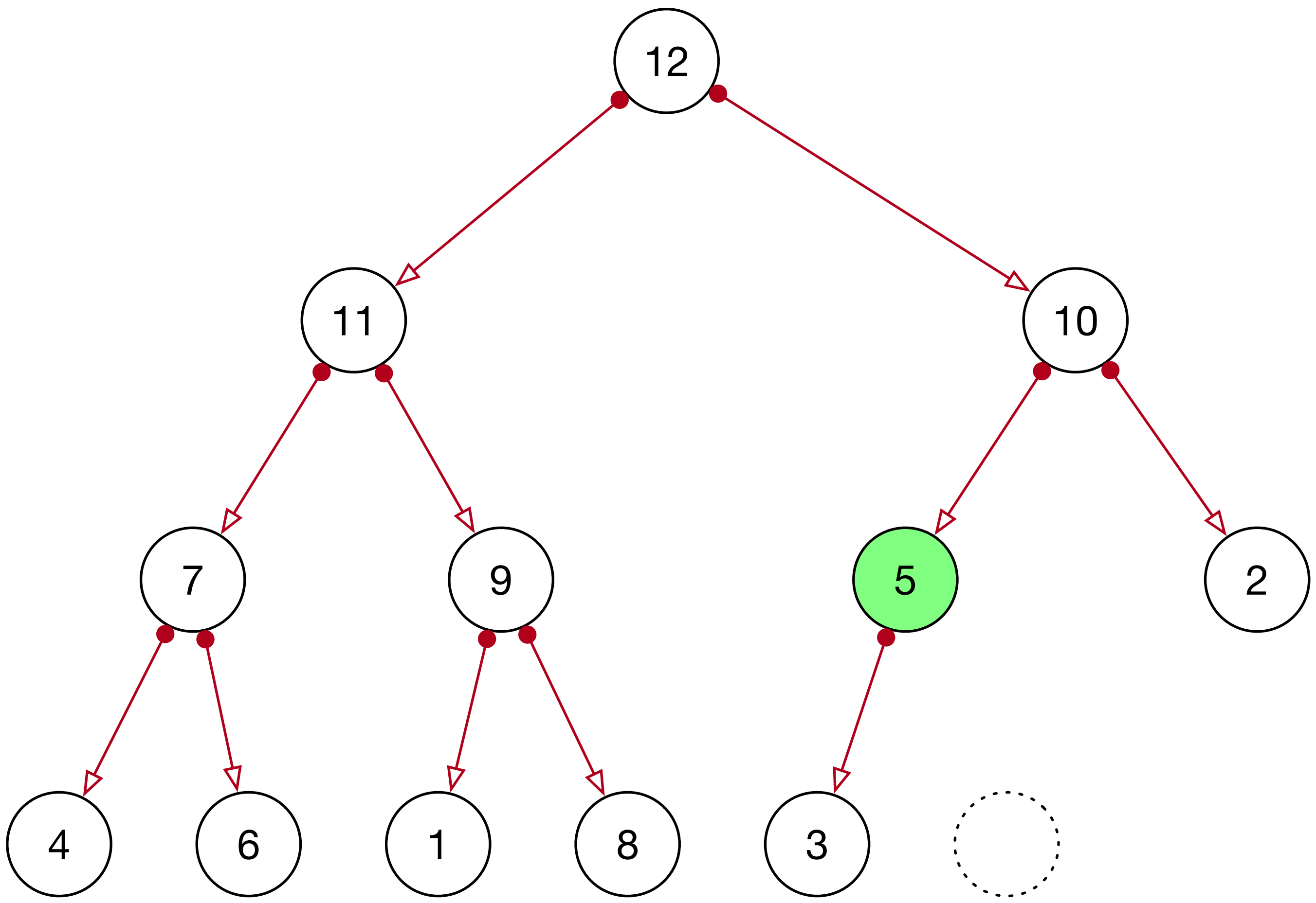
Priority Queues

- Then restore the heap property
  - Move up the ***larger*** sibling



Priority Queues

- Until there is no violation



Priority Queues

- Implementation:
    - Need to implement two "heapify" operations
        - Going up for insert
        - Going down for extract maximum

Priority Queues

- Define a class PQ with class methods for index calculation

```
class PQ:
    def __init__(self):
        self.array = []
    def up(index):
        return (index+1)//2-1
    def left(index):
        return 2*index + 1
    def right(index):
        return 2*index + 2
```

Priority Queues

- Insert at the end of the array
  - but note the index

```
def insert(self, value):
    n = len(self.array)
    self.array.append(value)
    while n>0:
        parent = PQ.up(n)
        print(n, parent, 'indices')
        if self.array[parent] < value:
            self.array[n], self.array[parent] =
               self.array[parent], self.array[n]
            n = parent
        else:
            return
```

Priority Queues

- Adjust by swapping with parent
  - Index of current element is *n*

```
def insert(self, value):
    n = len(self.array)
    self.array.append(value)
    while n>0:
        parent = PQ.up(n)
        print(n, parent, 'indices')
        if self.array[parent] < value:
            self.array[n], self.array[parent] =
               self.array[parent], self.array[n]
            n = parent
        else:
            return
```

Priority Queues

- Calculate the parent node

```
def insert(self, value):
    n = len(self.array)
    self.array.append(value)
    while n>0:
        parent = PQ.up(n)

        if self.array[parent] < value:
            self.array[n], self.array[parent] =
               self.array[parent], self.array[n]
            n = parent
        else:
            return
```

Priority Queues

- And swap if necessary

```
def insert(self, value):
    n = len(self.array)
    self.array.append(value)
    while n>0:
        parent = PQ.up(n)

        if self.array[parent] < value:
            self.array[n], self.array[parent] =
               self.array[parent], self.array[n]
            n = parent
        else:
            return
```

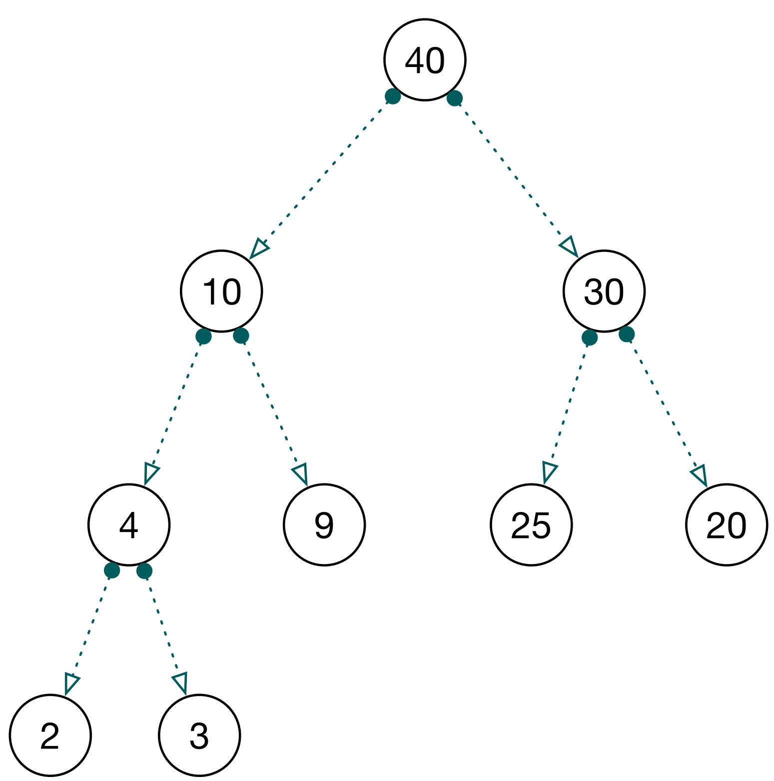
Priority Queues

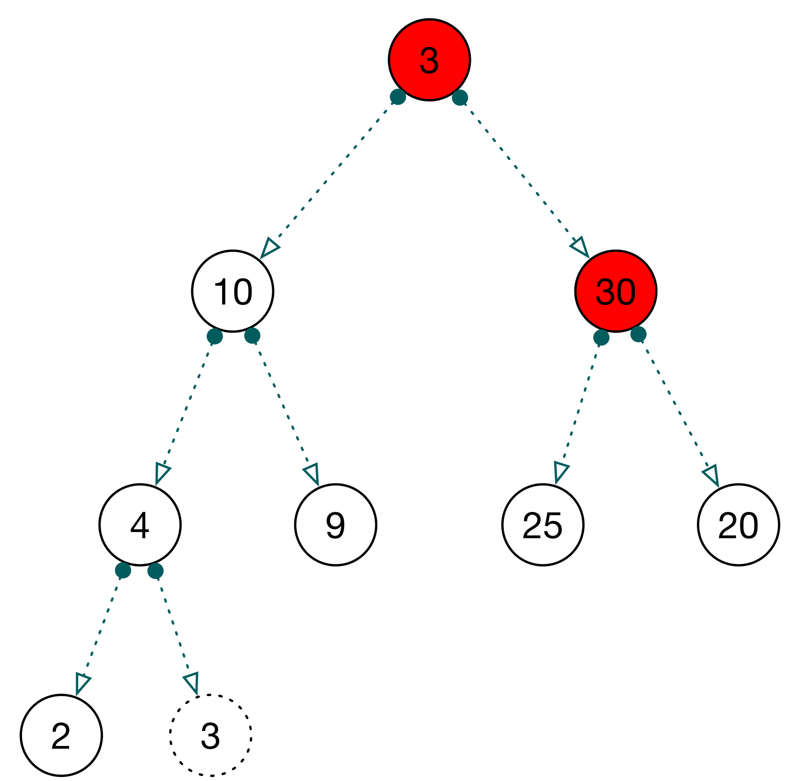
- Then reset the index

```
def insert(self, value):
    n = len(self.array)
    self.array.append(value)
    while n>0:
        parent = PQ.up(n)
        print(n, parent, 'indices')
        if self.array[parent] < value:
            self.array[n], self.array[parent] =
               self.array[parent], self.array[n]
            n = parent
        else:
            return
```

Priority Queues

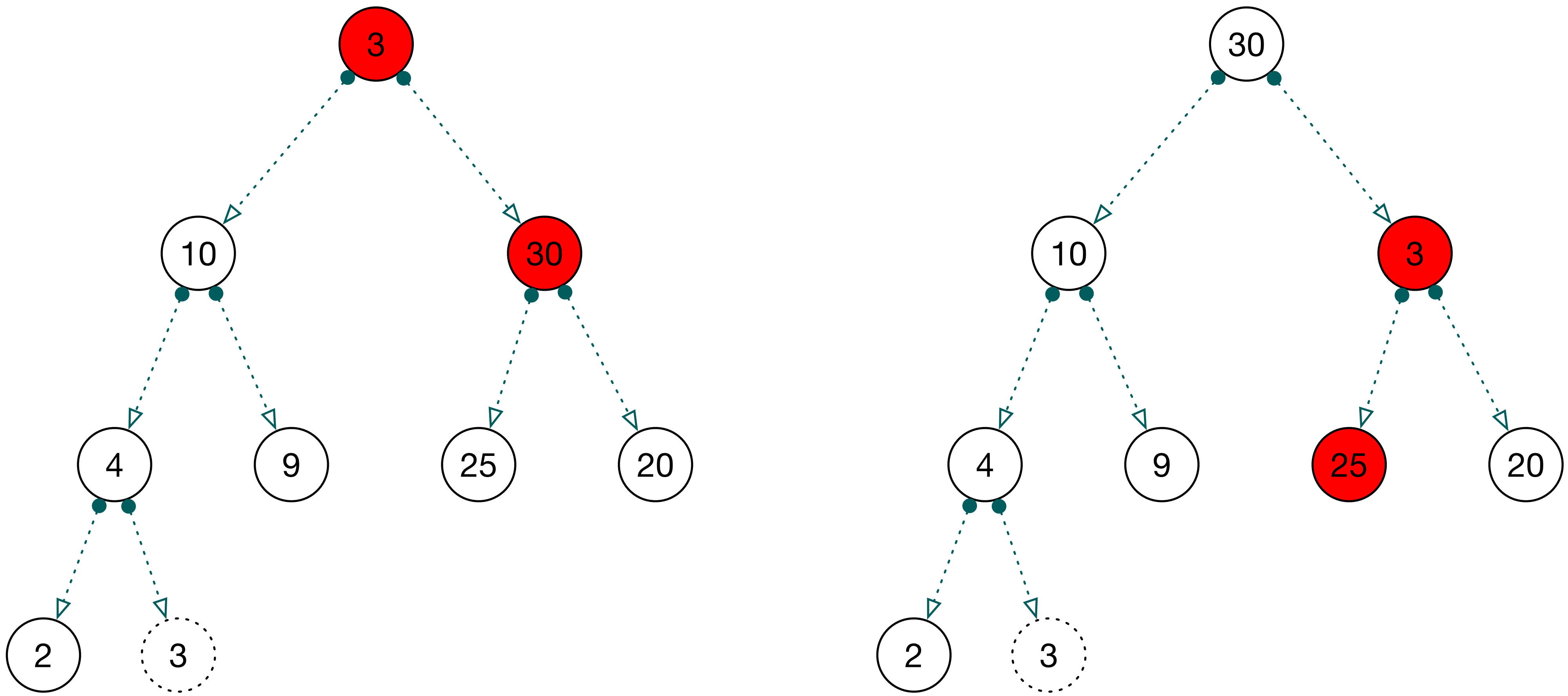
- Extract maximum:
  - Maximum is always at position 0
  - Swap its value with the last element in the array
  - Then heapify:



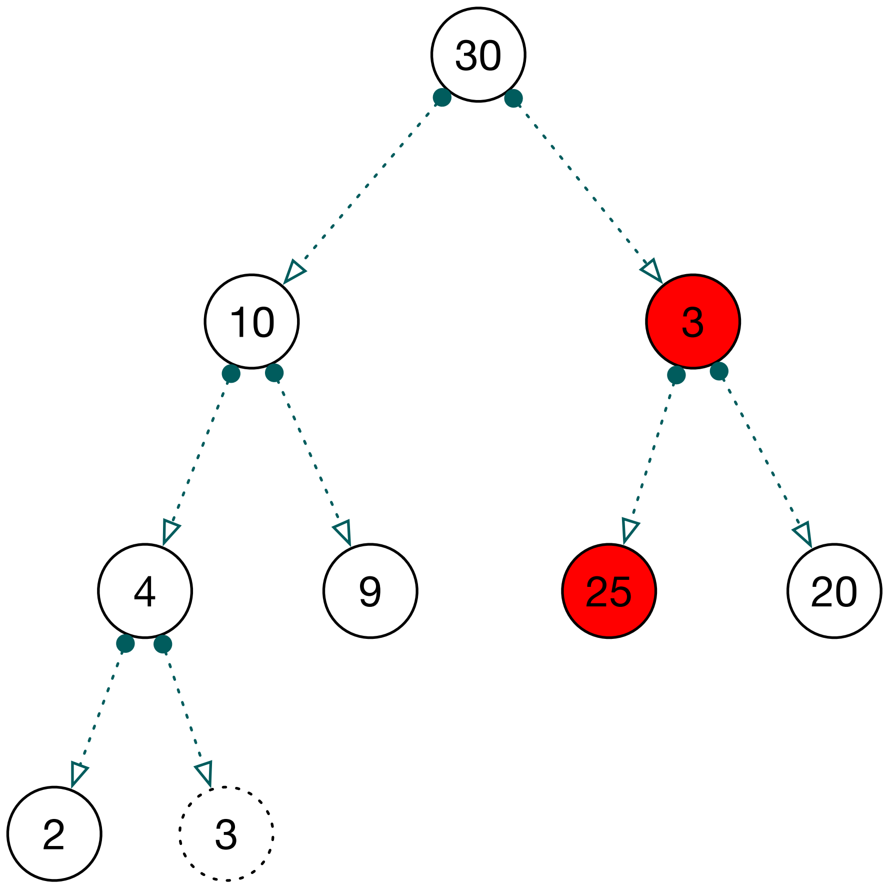


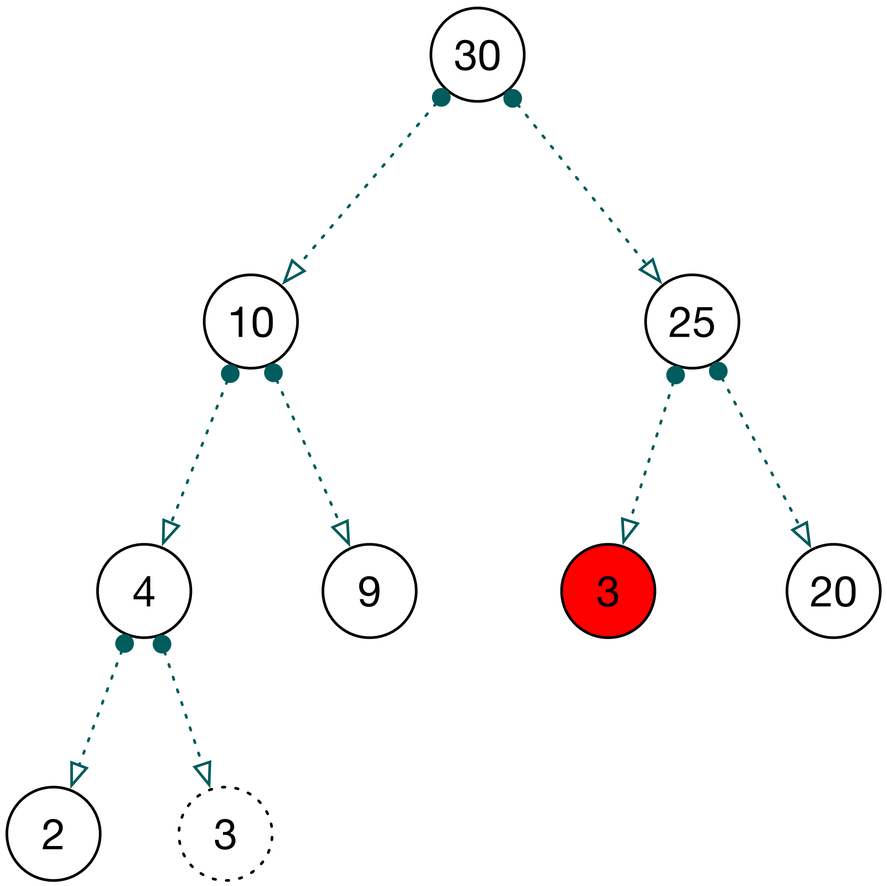
Priority Queues

- This is also recursive, but proceeds from top to bottom



Priority Queues





Priority Queues

- Swap last and first node
- Delete from node

```
def get_max(self):
     ret_val = self.array[0]
     last = self.array[-1]
     del self.array[-1]
     self.array[0] = last
     n=0
```

Priority Queues

- Now recursively recover the heap property
  - Make case distinctions according to whether
    - both children exist
    - only the left child exist
    - no children present

Priority Queues

- Both children exist

```
 def get_max(self):
     ...
     while n < len(self.array):
          left = PQ.left(n)
          right = PQ.right(n)
          if right < len(self.array):
              if self.array[n] > self.array[left] and
                        self.array[n] > self.array[right]:
                  return ret_val
              if self.array[left] < self.array[right]:
                  m = right
              else:
                  m = left
              self.array[n], self.array[m] = self.array[m],
self.array[n]
              n = m
```

Priority Queues

- Heap property is not violated

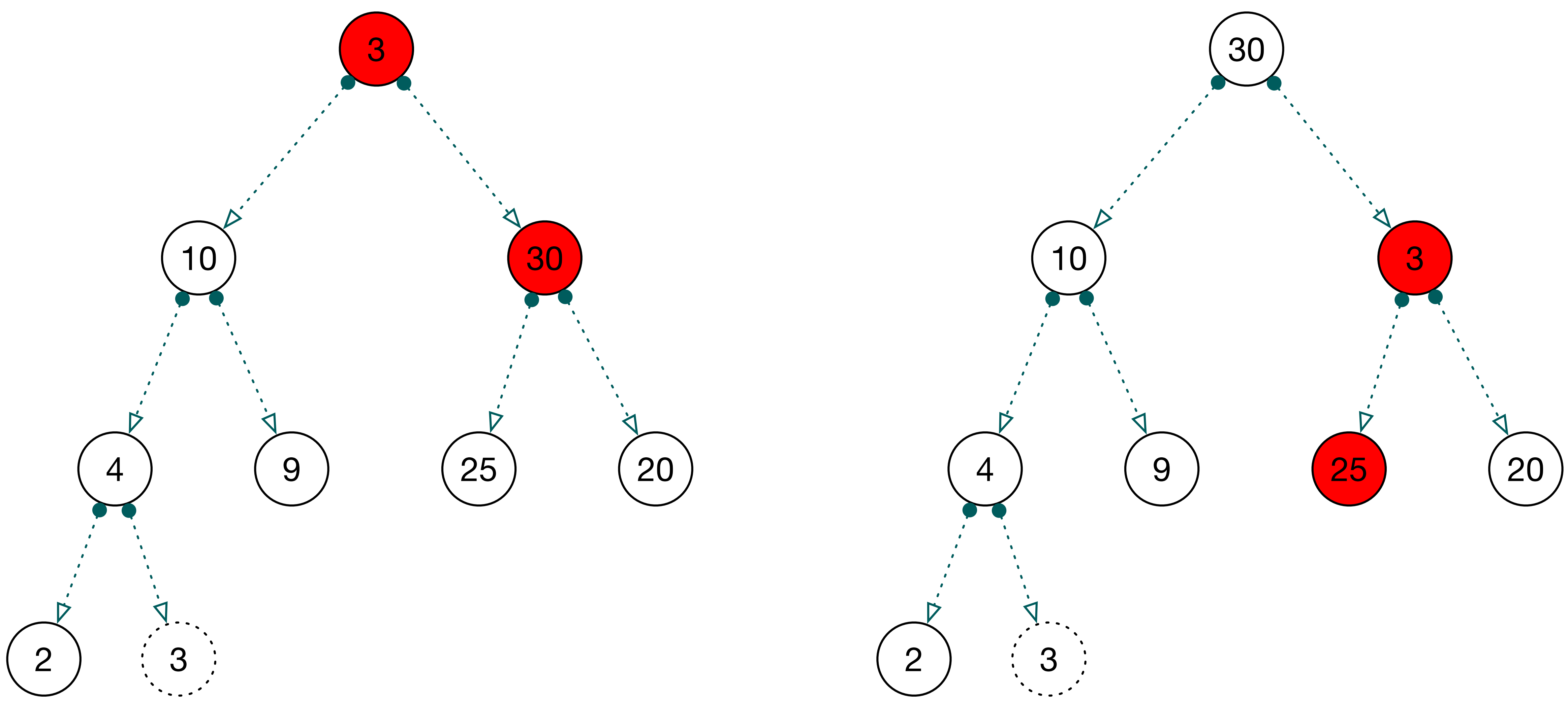
```
def get_max(self):
    ...
    while n < len(self.array):
        left = PQ.left(n)
        right = PQ.right(n)
        if right < len(self.array):
            if self.array[n] > self.array[left] and
                    self.array[n] > self.array[right]:
                return ret_val
            if self.array[left] < self.array[right]:
                m = right
            else:
                m = left
            self.array[n], self.array[m] = self.array[m],
self.array[n]
            n = m
```

Priority Queues

- Select the larger of the two children for swapping

```
def get_max(self):
    ...
    while n < len(self.array):
        left = PQ.left(n)
        right = PQ.right(n)
        if right < len(self.array):
            if self.array[n] > self.array[left] and
                      self.array[n] > self.array[right]:
                return ret_val
            if self.array[left] < self.array[right]:
                m = right
            else:
                m = left
            self.array[n], self.array[m] =
                  self.array[m], self.array[n]
            n = m
```

Priority Queues



Priority Queues

- Swap

```
def get_max(self):
    ...
    while n < len(self.array):
        left = PQ.left(n)
        right = PQ.right(n)
        if right < len(self.array):
            if self.array[n] > self.array[left] and
                      self.array[n] > self.array[right]:
                return ret_val
            if self.array[left] < self.array[right]:
                m = right
            else:
                m = left
            self.array[n], self.array[m] =
                  self.array[m], self.array[n]
            n = m
```

Priority Queues

- Swap

```
def get_max(self):
   ...
   while n < len(self.array):
        left = PQ.left(n)
        right = PQ.right(n)
        if right < len(self.array):
            if self.array[n] > self.array[left] and
                      self.array[n] > self.array[right]:
                return ret_val
            if self.array[left] < self.array[right]:
                m = right
            else:
                m = left
            self.array[n], self.array[m] =
                  self.array[m], self.array[n]
            n = m
```

Priority Queues

- And do not forget to set yourself up for recursion

```
def get_max(self):
   ...
   while n < len(self.array):
        left = PQ.left(n)
        right = PQ.right(n)
        if right < len(self.array):
            if self.array[n] > self.array[left] and
                      self.array[n] > self.array[right]:
                return ret_val
            if self.array[left] < self.array[right]:
                m = right
            else:
                m = left
            self.array[n], self.array[m] =
                  self.array[m], self.array[n]
            n = m
```

Priority Queues

- Only one child can exist (but then it has to be the left one)
    - Heap property might not be violated

```
elif left < len(self.array):
    if self.array[n] > self.array[left]:
        return ret_val
    m = left
    self.array[n], self.array[m] =
              self.array[m], self.array[n]
    n = m
```

Priority Queues

- Only one child can exist (but then it has to be the left one)
  - But if it is, we have only one candidate for swapping

```
elif left < len(self.array):
     if self.array[n] > self.array[left]:
         return ret_val
     m = left
     self.array[n], self.array[m] =
               self.array[m], self.array[n]
     n = m
```

Priority Queues

- Per defensive programming, we pretend that we might have to go on:

```
elif left < len(self.array):
     if self.array[n] > self.array[left]:
          return ret_val
     m = left
     self.array[n], self.array[m] =
               self.array[m], self.array[n]
     n = m
```

Priority Queues

- Difficult Homework:
  - Extract Maximum and insertion of a new element are sometimes combined
  - In this case, we can save work by:
    - inserting the new element at the beginning of the array
    - work ourselves downwards to restore the heap property
  - Implement this

Priority Queues

- Other operations:
  - peek
    - returns the maximum, but does not remove it
  - is_empty
    - checks whether the array is empty

Priority Queues

- Costs of operations
  - Priority queue with n elements uses $\log_2(n)$ steps in order to heapify
  - Peek and is_empty run in constant time

Priority Queues

- Python implementation of priority queues
  - heapq implements a minimum heap
  - Uses a Python list

```
heapq.heappush(lista, element)

heapq.heappop(lista)
```

Priority Queues

- This is an efficient implementation
    - We can "kludge" a max heap implementation for integers by observing that the maximum of numbers is the negative of the negative integers

```
def smallpush(lista, element):
    heapq.heappush(lista, -element)
def smallpop(lista):
    return -heapq.heappop(lista)
```

Running Medians

- Task:
  - We are given a stream of numbers
    - At any time, want to be able to determine the median of these numbers
- Example:
  - We get 5, 3, 1, 10, 2
  - Median is now 3
  - We then get 12, 1, 2
    - We have seen 1,1,2,2,3,5,10,12
  - Median is now 2.5 (mean of 2 and 3)

Running Medians

- Naïve implementation
  - Just keep an ordered list around
- Better way:
  - Keep two sublists of equal size
    - Small and Big
    - All elements in Small are smaller than all elements in Big
    - Use heaps in order to easily extract the maximum of Small and the minimum of Big

Running Medians

- Adding a new number:
  - If the left heap is smaller, then insert there
  - If the left and right heap have equal size, insert in the right heap
    - But need to maintain the invariant:
      - All elements in the left heap are smaller (or equal) than all elements in the right heap

Running Medians

- Example: Inserting 5 into
  - Left: 0, 1, 1, 2, 2 Right: 3, 4, 6, 7, 7, 9
- We need to insert into Left, but this violates the invariant
  - Extract the minimum from right (3)
  - Add the minimum to the left
  - Add 5 to right
  - Left: 0, 1, 1, 2, 2, 3 Right: 4, 5, 6, 7, 7, 9

Running Medians

- Insert another 5:
  - Left: 0, 1, 1, 2, 2, 3   Right: 4, 5, 6, 7, 7, 9
- Rule say insert to the Right:
  - Since max(left) < 5:
    - No problem:
  - Left: 0, 1, 1, 2, 2, 3   Right: 4, 5, 5, 6, 7, 7, 9

Running Medians

- Insert another 5:
  - Insert into Left:
    - But min(right) = 4 which is smaller than 5
  - Inserting 5 into left violates the invariant
    - Need to do something about it:
      - Extract minimum from Right
      - Insert this minimum into Left
      - Insert new element into Right
  - Left: 0, 1, 1, 2, 2, 3, 4   Right: 5, 5, 6, 7, 7, 9

Running Medians

- Calculating medians:
  - If len(Left) < len(Right):
    - Median is peek(Right)
  - Otherwise:
    - Median is (peek(Right)+peek(Left))/2

Tiered Bitvectors

- Given a finite universe indexed by $\{0,1,\ldots,n-1\}$:
  - $U = \{x_i : i \in \{0,1,\ldots,n-1\}\}$
- A bit vector represents subsets of U
  - If $\delta_{i,S} = \begin{cases} 0 & \text{if } i \in S \\ 1 & \text{if } i \notin S \end{cases}$ (Kronecker delta)
  - $S \subset U$ corresponds to the bit-vector
    - $(\delta_{i,S} : i \in \{0,1,\ldots,n-1\})$

Tiered Bitvectors

- To insert x_i into the set:
  - Set bit i to 1
- To delete x_i from the set:
  - Set bit i to 0
- To lookup whether $x_i \in S$:
  - Check value of bit i

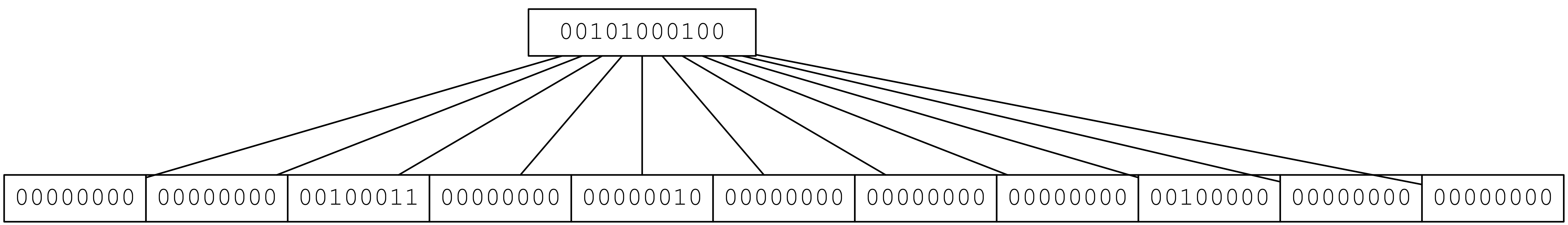
Tiered Bitvectors

- MinIndex, MaxIndex, PredecessorIndex, SuccessorIndex are extremely slow
  - $\Theta(|U|)$

Tiered Bitvectors

- If memory is (as is typical) accessed via cache-lines
  - Words of length $L = 64$B
- Break the universe into $|U|/L$ pieces of size L
  - $U_0, U_1, U_2, \ldots, U_{n/L}$
- Introduce a master bitvector by
  - $\Delta_{i,S} = \begin{cases} 0 & \text{if } S \cap U_i = \emptyset \\ 1 & \text{if } S \cap U_i \neq \emptyset \end{cases}$

Tiered Bitvectors



Tiered Bitvectors

- Ordered dictionary operations run in time
  - $O(U/L + L)$
- This is minimized when $L = \sqrt{U}$

Tiered Bitvectors

- What are the operations?

Tiered Bitvectors

- Inserting to a set:
  - Set master bitvector bit
  - Set partial bitvector bit
- Deleting from a set
  - Reset partial bitvector bit
  - If partial bitvector is empty: Reset master bitvector bit

Tiered Bitvectors

- is-empty:
  - Check master bitvector only has entries 0
- min:
  - One min operation on the master bitvector, one min operation on partial bitvector

Tiered Bitvectors

- Obviously, we can extend this from two tiers to many tiers.
  - Result is a tree
  -

Fibonacci Heaps

Mergeable Heaps

- Mergeable Heaps are ADS with
  - Make-Heap
  - Insert(*H*, *x*)
  - Minimum(*H*)
  - Extract-Min(*H*)
  - Union(H_1, H_2)
- Fibonacci heaps in addition have
  - Decrease-Key(*H, x, new_val*)
  - Delete(*H,x*)

Mergeable Heaps

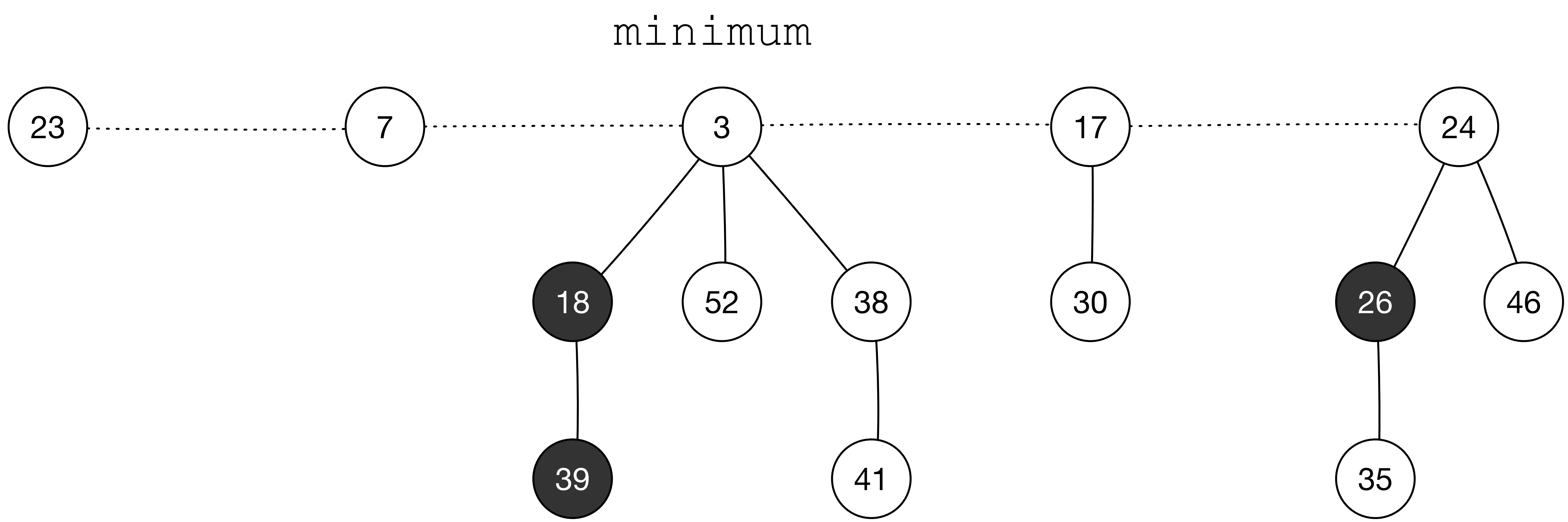
Procedure	Binary Heap (worst-case)	Fibonacci Heap (amortized)
Make-Heap	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\log n)$	$\Theta(1)$
Minimum	$\Theta(1)$	$\Theta(1)$
Extract-Min	$\Theta(\log n)$	$O(\log n)$
Union	$\Theta(n)$	$\Theta(1)$
Decrease-key	$\Theta(\log n)$	$\Theta(1)$
Delete	$\Theta(\log n)$	$O(\log n)$

Fibonacci Heaps

- Fibonacci heaps:
  - Useful when Extract-Min and Delete are rare
  - E.g. graph algorithms where we use decrease-key in order to update edges
    - Minimum spanning trees
    - Single-source shortest paths
  -

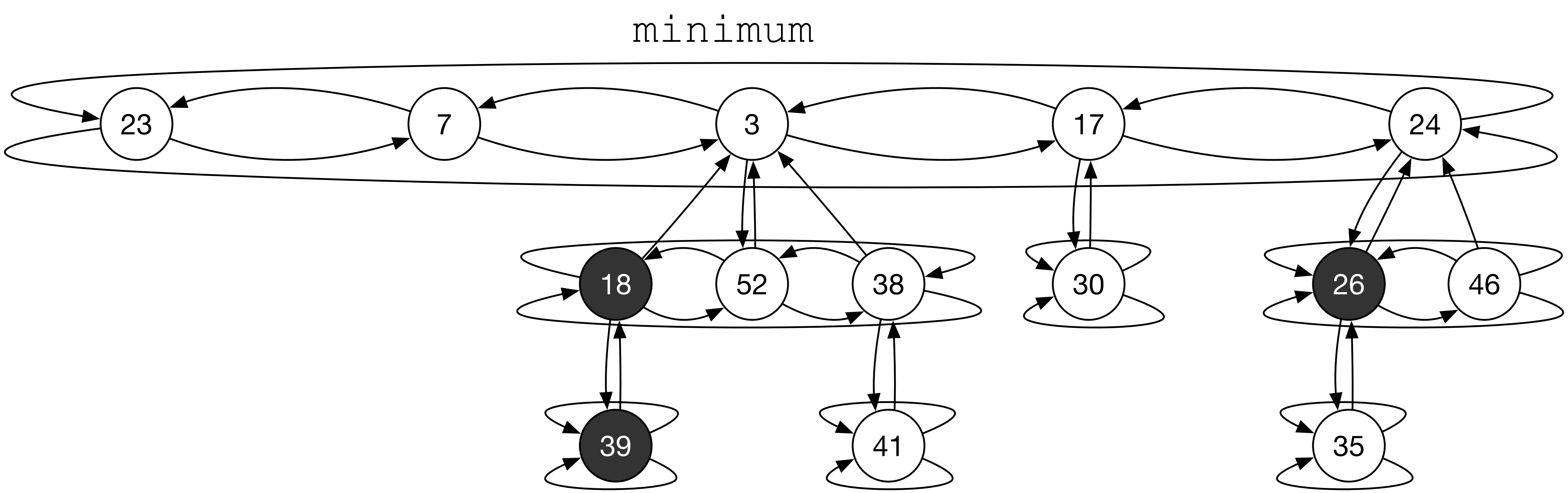
Fibonacci Heaps

- Fibonacci heap:
  - Collection of rooted trees that are min-heap ordered:
    - Every child's key is larger than its parent's key
- Example:



Fibonacci Heaps

- Each node has a link to their parent
- All children (including root list) are in a double linked child list
- Parent has one link to the child list



Fibonacci Heaps

- Double linked list:
  - Allows constant time inserts and fusion

Fibonacci Heaps

- Each node has attributes
  - Number of children in degree
  - Node x has mark x.mark to indicate whether node x has lost a child since the last time x was made a child of another node
    - Newly created nodes are unmarked
    - Node x becomes unmarked whenever it is made child of another node
    - Used for DecreaseKey operation

Fibonacci Heaps

- Access to a Fibonacci node through pointer to minimum key node
- Each Fibonacci node stores
  - Links to left and right sibling
  - Link to parent unless root
  - Link into the child list

Fibonacci Heaps

- Use potential method
  - $t(H)$: number of trees in the root list
  - $m(H)$: number of marked nodes
- Potential $\Phi(H) = t(H) + 2m(H)$
  - Unit of potential can pay for all constant time operations

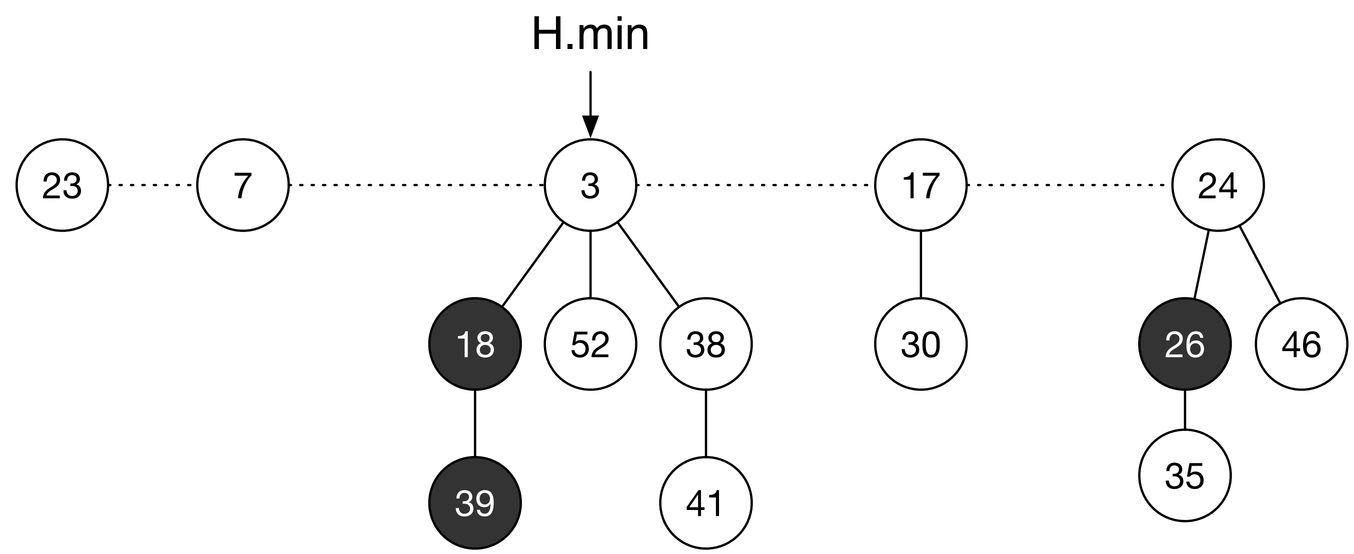
Fibonacci Heap Operations

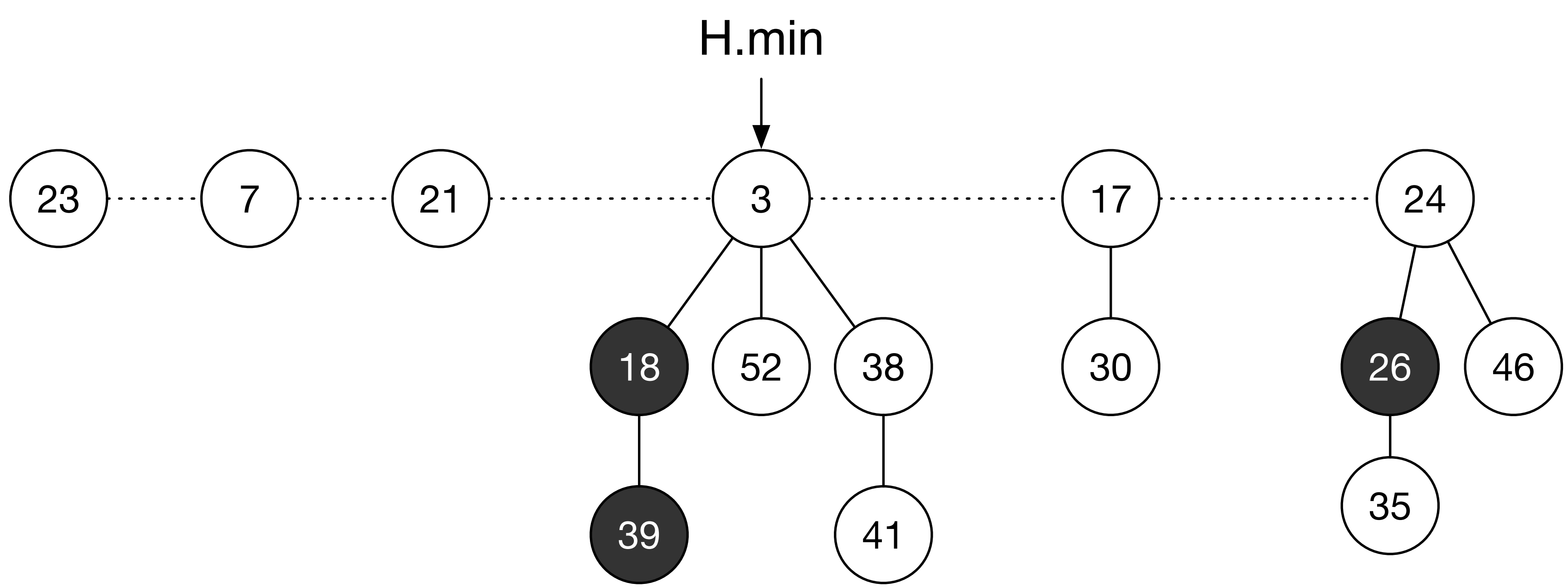
- Creating a new Fibonacci Heap
  - $H.n = 0$
  - $H.\min =$ Null
  - $\Phi(H) = 0$
  - amortized cost is $O(1)$

Fibonacci Heaps

- Insert(H,x)
  - Create an otherwise empty tree with x as root
  - If there is no element in the heap (H.min = Null):
    - Create a root list for H containing x only
  - Otherwise
    - Insert the new tree into the root list
    - Check whether H.min needs to be updated
- Amortized costs: $O(1) + 1 = O(1)$

Fibonacci Heaps





Fibonacci Heaps

- Minimum
    - Just returns a pointer to the minimum node
- Amortized costs is $O(1)$

Fibonacci Heaps

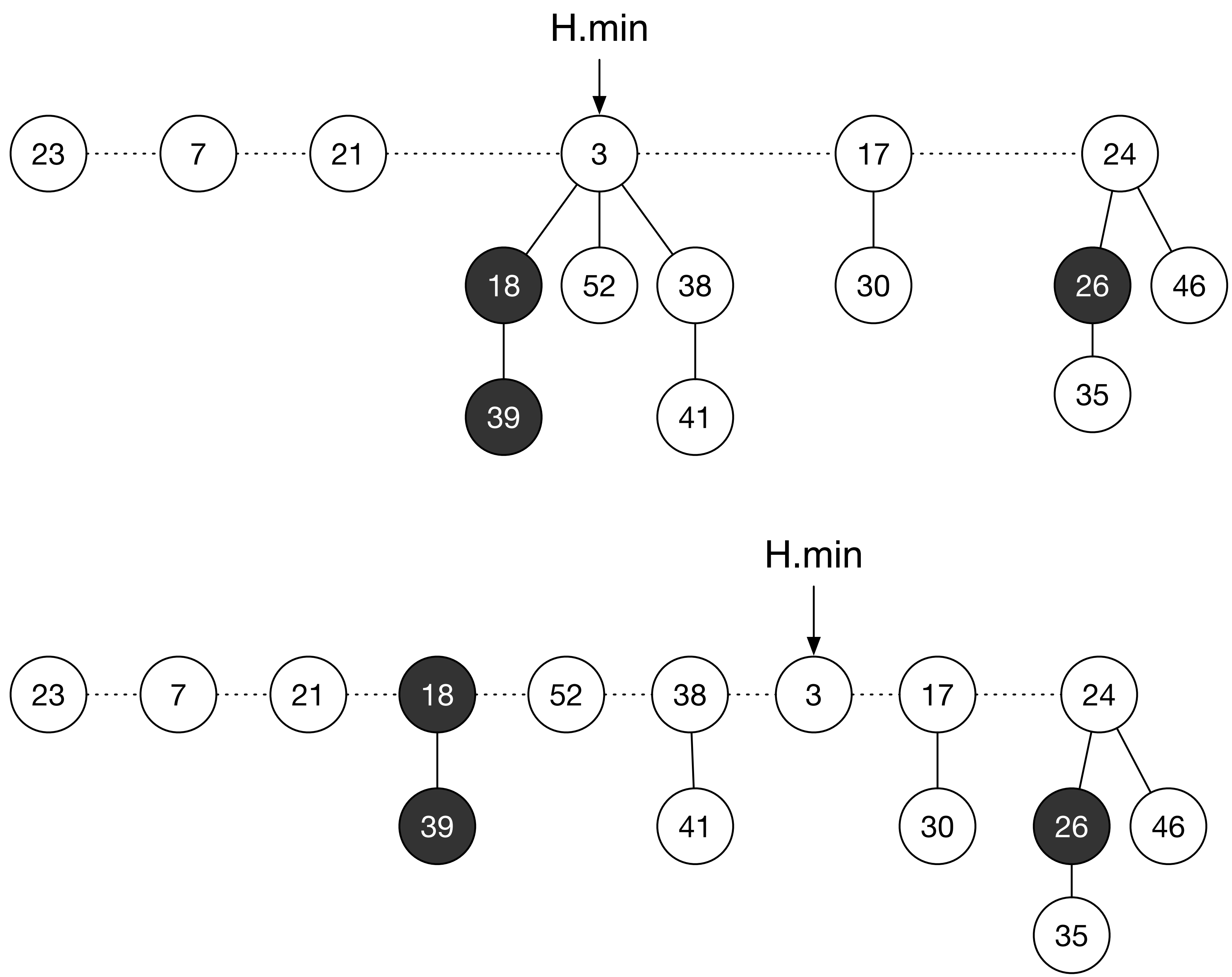
- Uniting two heaps
  - Concatenate the root lists
- Change in potential is zero
- Amortized costs is $O(1)$

Fibonacci Heaps

- Extracting the minimum
  - This is where we consolidate
  -

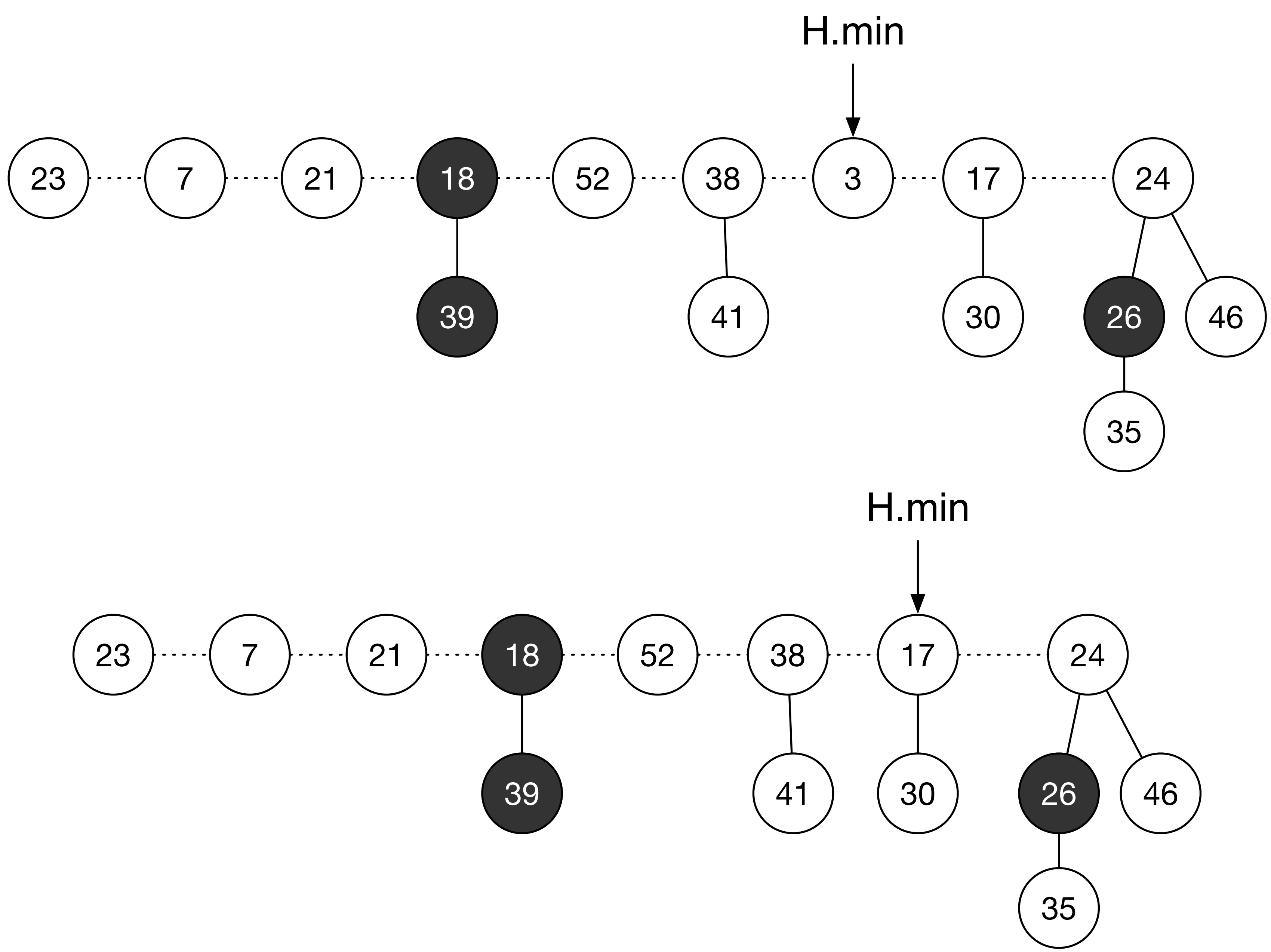
```
z=H.min
if z ≠ NULL:
   for each child x of z:
       add x to the root list of H
       x.p = NULL
    remove z from root list
    if z == z.right: #z only node
        H.min = NULL
    else:
        H.min = z.right
        CONSOLIDATE(H)
    H.n -= 1
return z
```

Fibonacci Heaps



Moving children of H.min into the root list

Fibonacci Heaps



Removing and returning H.min

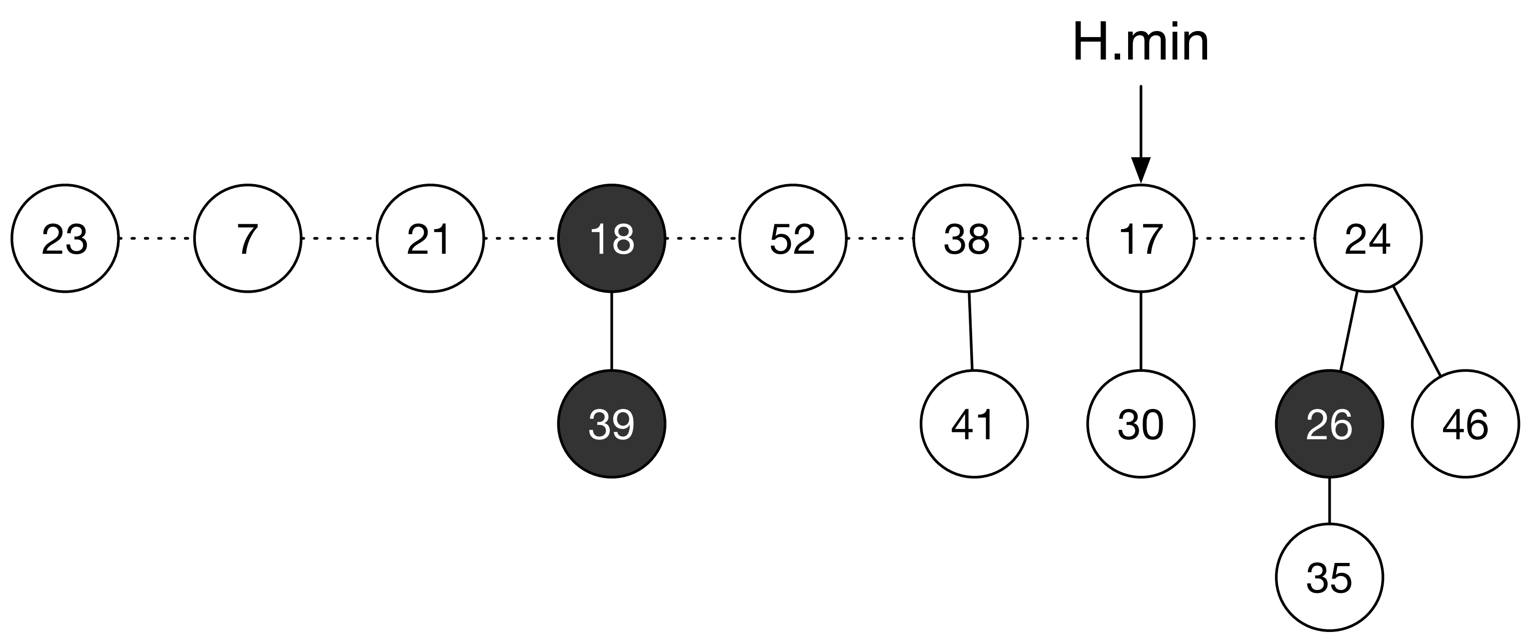
Fibonacci Heaps

- Consolidation:
  - Repeat:
    - Find two nodes in the root list with trees of the same height
      - Starting at the current link H.min
    - Unify the two trees making the smaller one the root
      - Clears mark on the looser

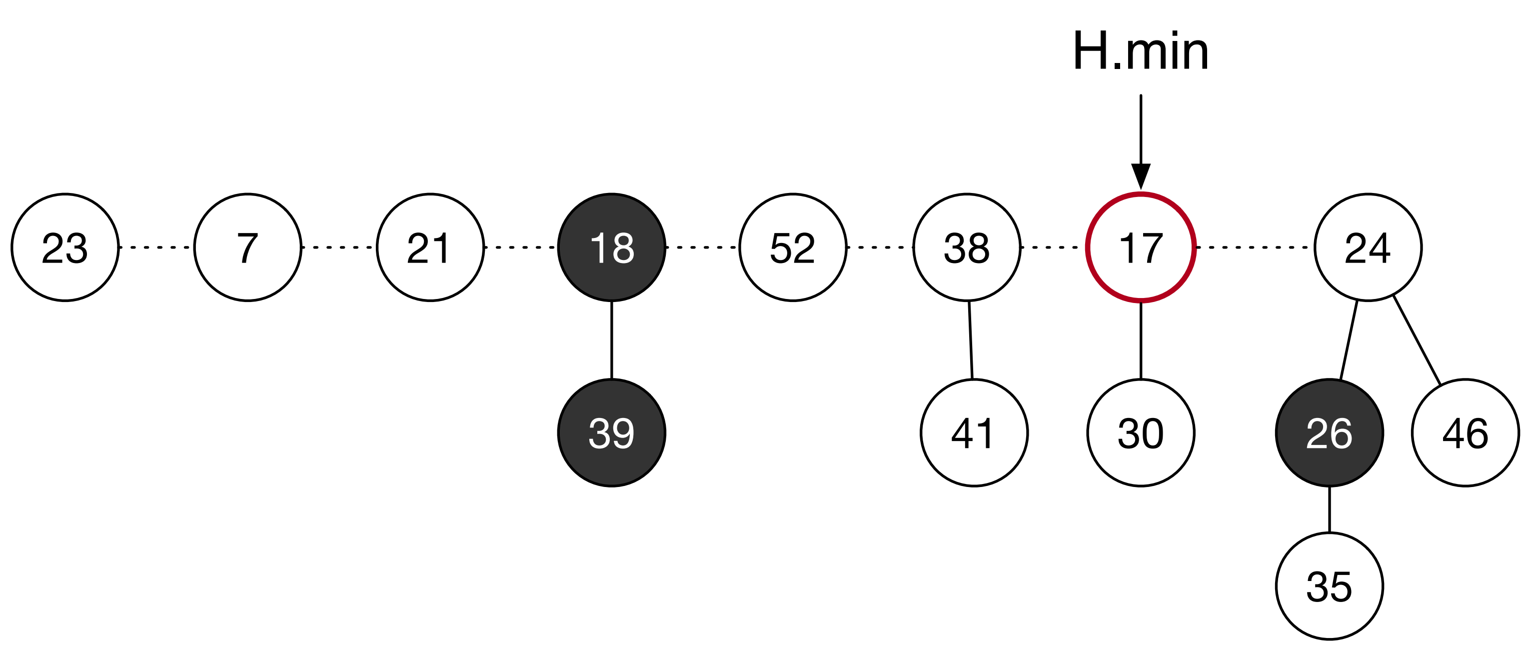
Fibonacci Heaps

- Consolidation:
  - To find nodes in the root list that can be merged
    - Create an array A[0 … D(H.n)] of nodes in the root tree
      - D(H.n) is the maximum degree of a tree rooted in a node in the root list
    - Fill into A
    -

Fibonacci Heaps

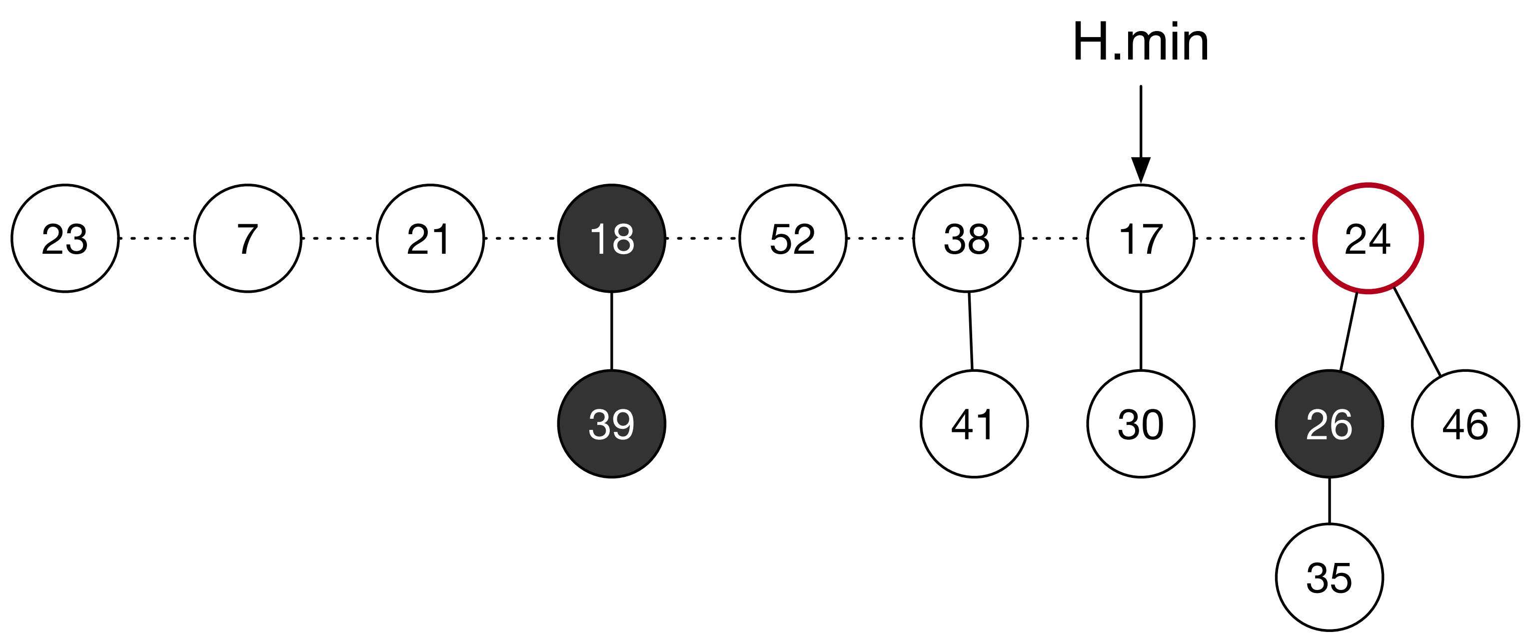


A[0]: NULL
A[1]: NULL
A[2]: NULL
A[3]: NULL



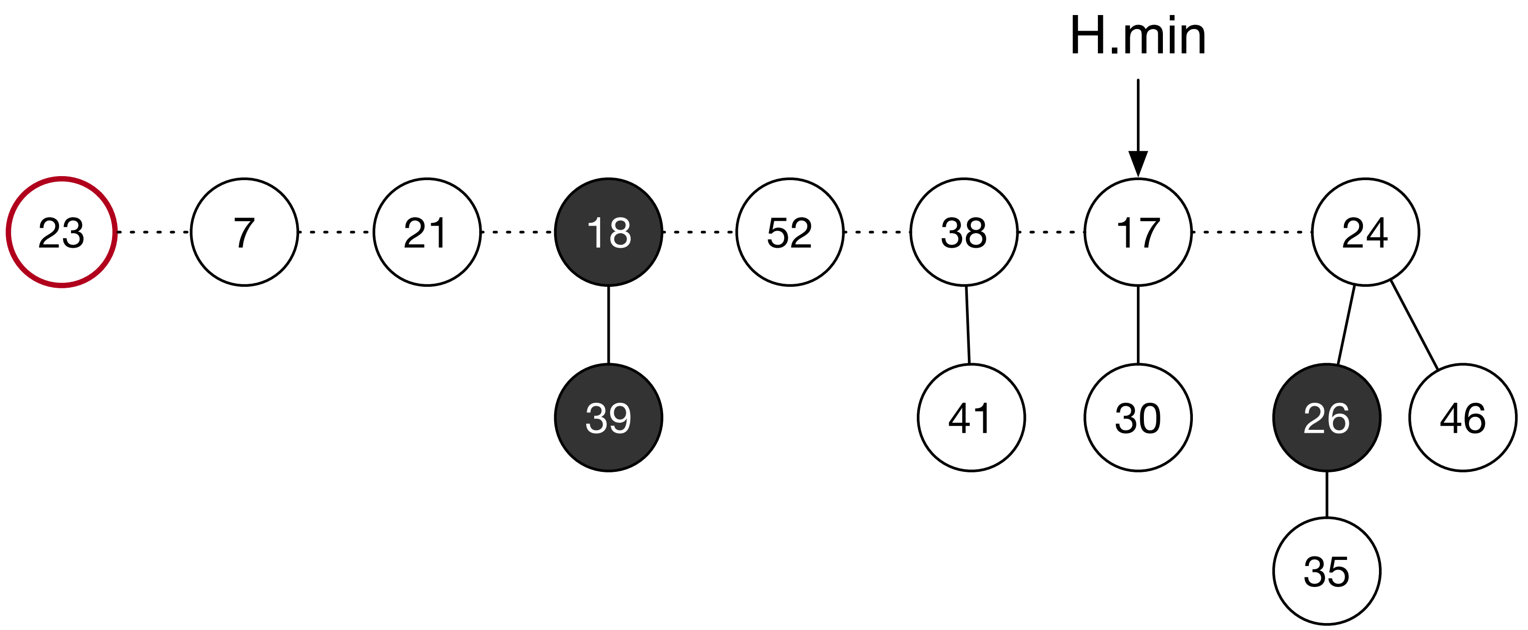
A[0]: NULL
A[1]: 17
A[2]: NULL
A[3]: NULL

Fibonacci Heaps



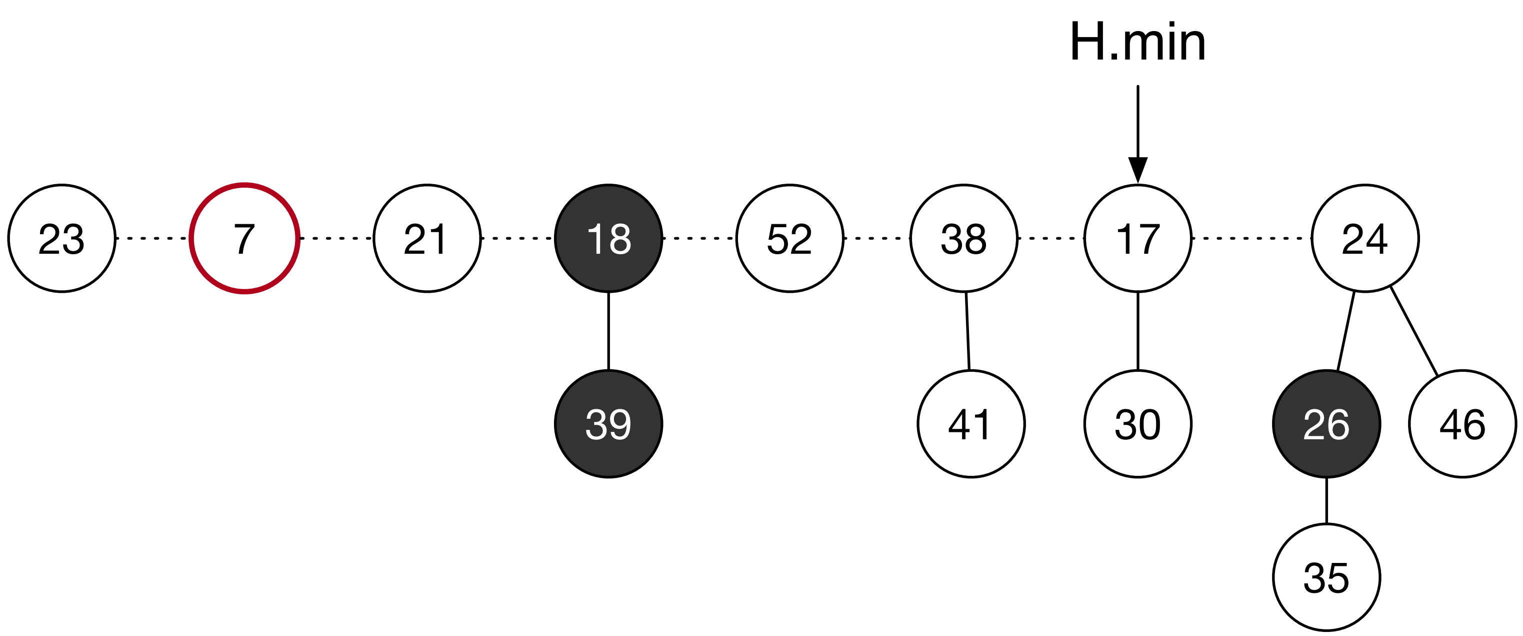
A[0]: NULL
A[1]: 17
A[2]: 24
A[3]: NULL

Fibonacci Heaps



A[0]: 23
A[1]: 17
A[2]: 24
A[3]: NULL

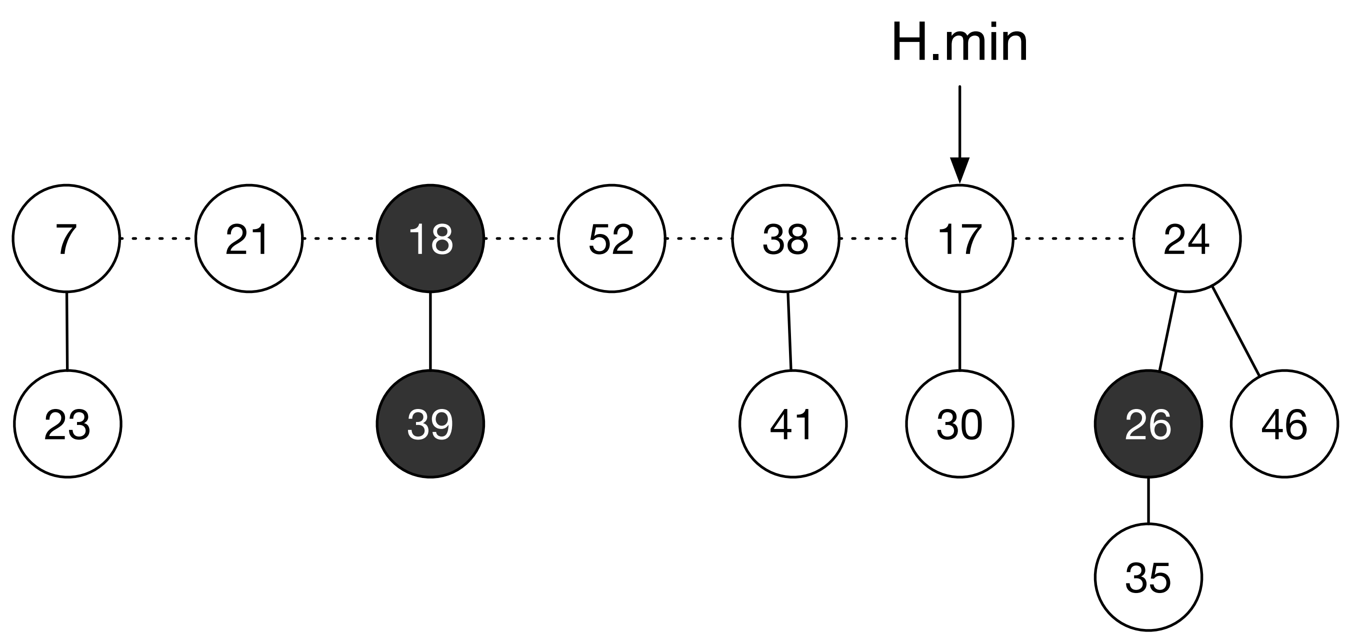
Fibonacci Heaps



A[0]: 23, 7
A[1]: 17
A[2]: 24
A[3]: NULL

We cannot insert 7 into A[0], so we combine

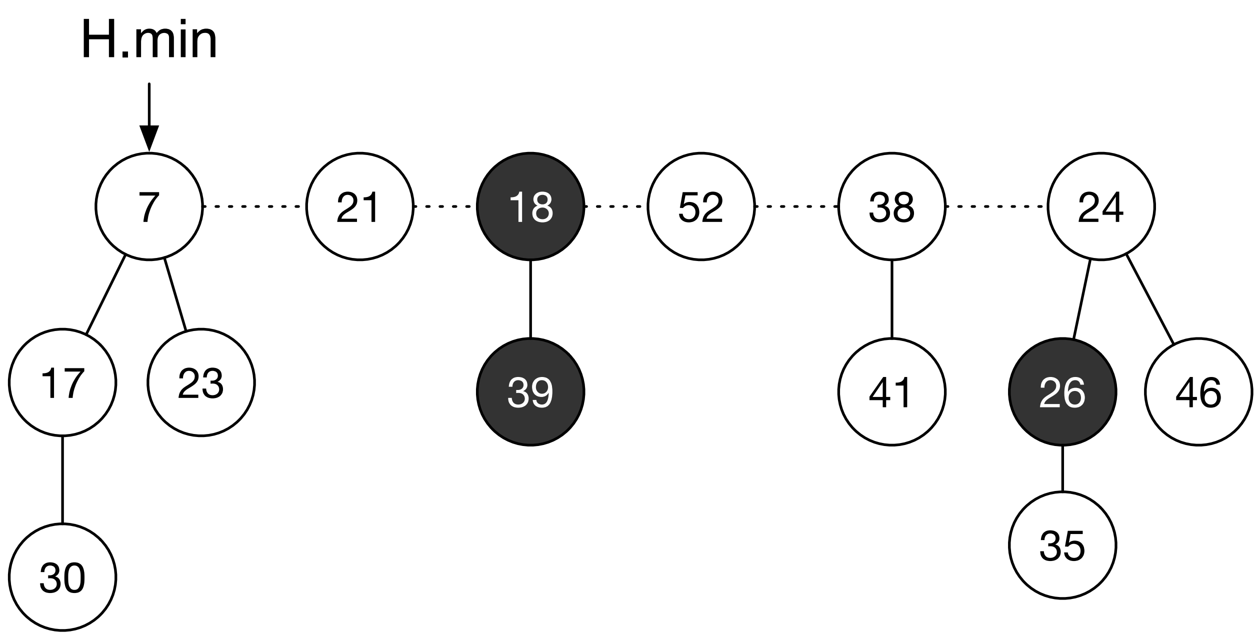
Fibonacci Heaps



A[0]: NULL
A[1]: 17, 7
A[2]: 24
A[3]: NULL

We remove 7 from the array. We then merge.
After merging, we try to insert 7 into A[1]. Since there is already an occupant there, we find another merge candidate

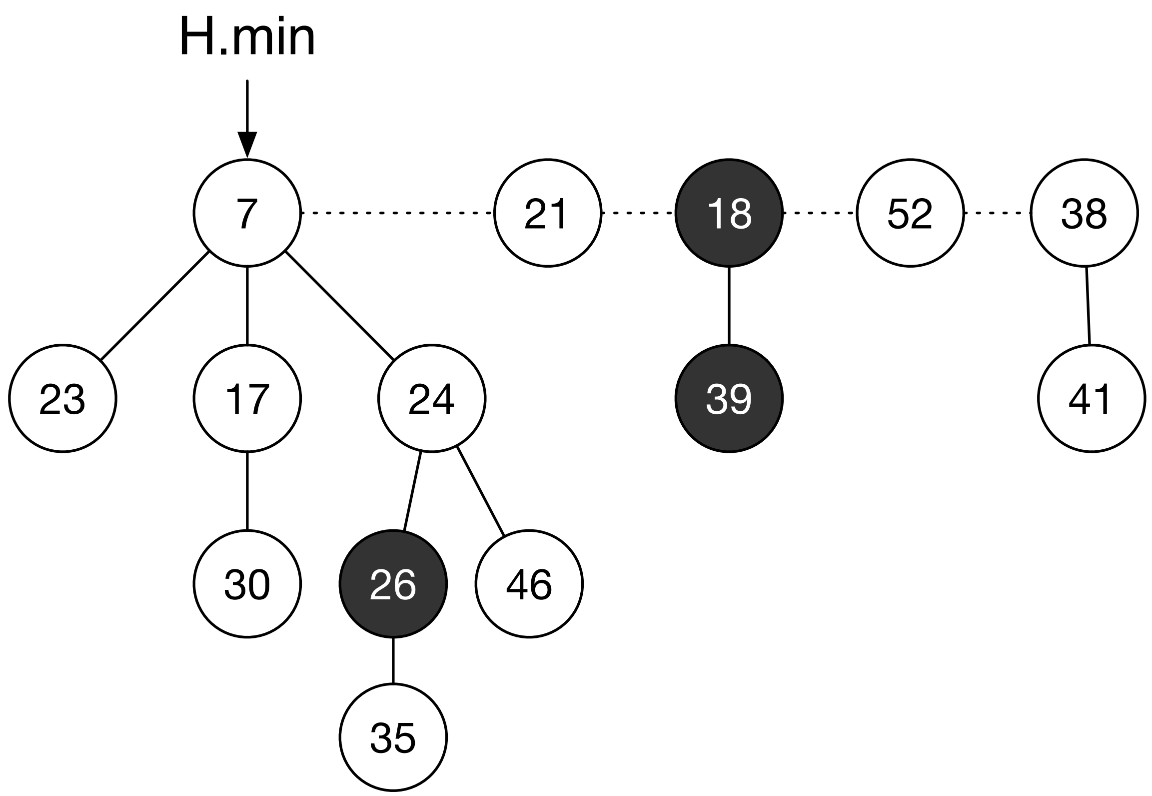
Fibonacci Heaps



A[0]: NULL
A[1]: NULL
A[2]: 24, 7
A[3]: NULL

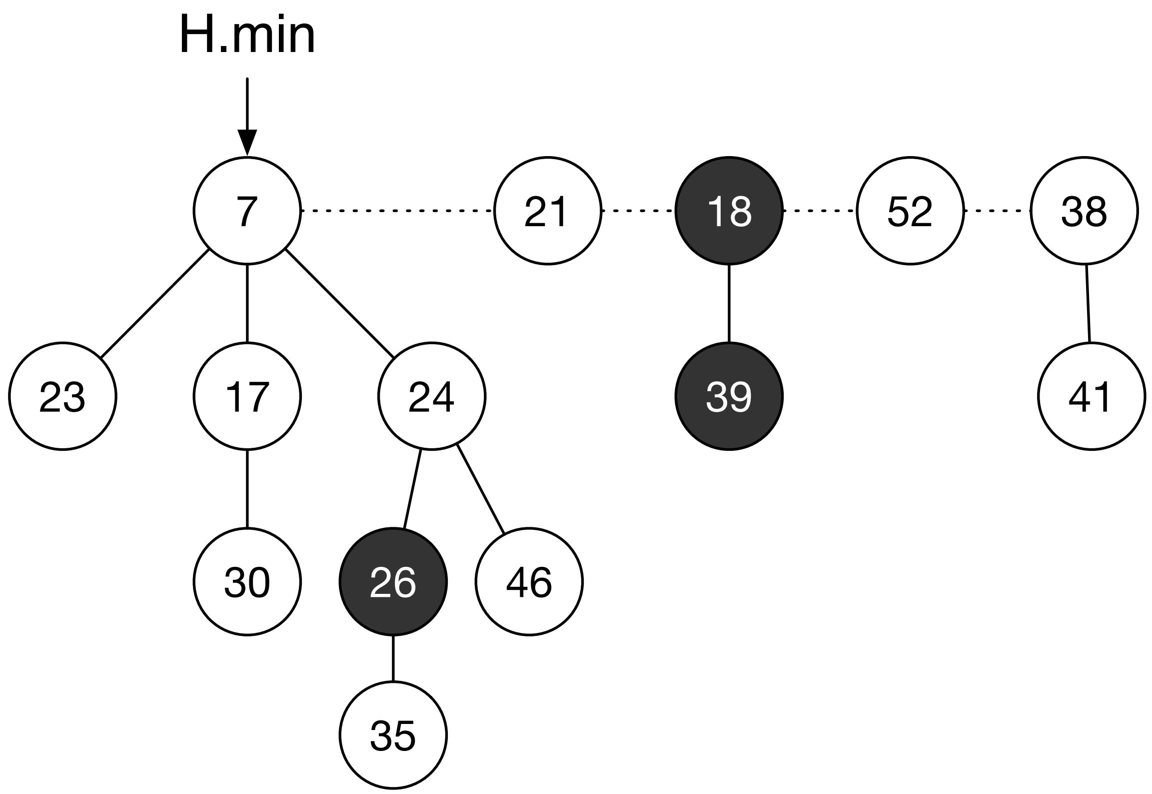
Inserting the new tree into A gives us another merge candidate

Fibonacci Heaps



A[0]: NULL
A[1]: NULL
A[2]: NULL
A[3]: 7

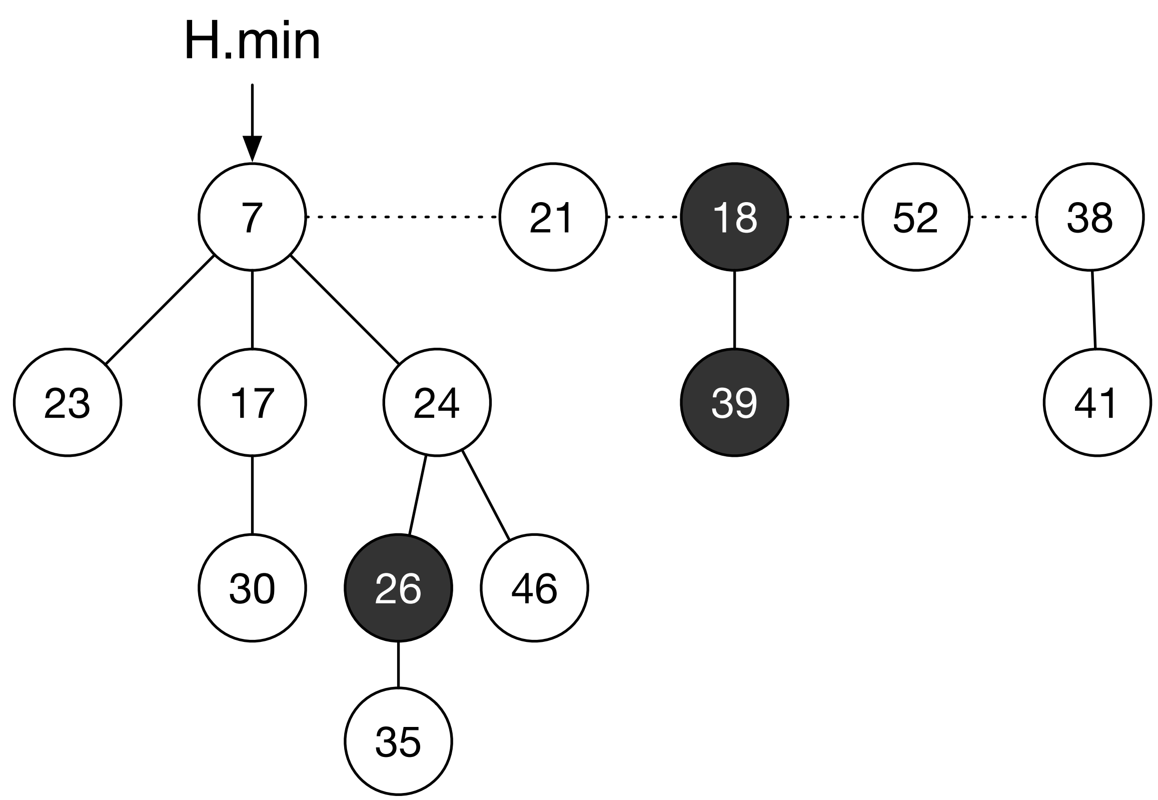
Fibonacci Heaps



A[0]: 21
A[1]: NULL
A[2]: NULL
A[3]: 7

We insert 21 into the array.

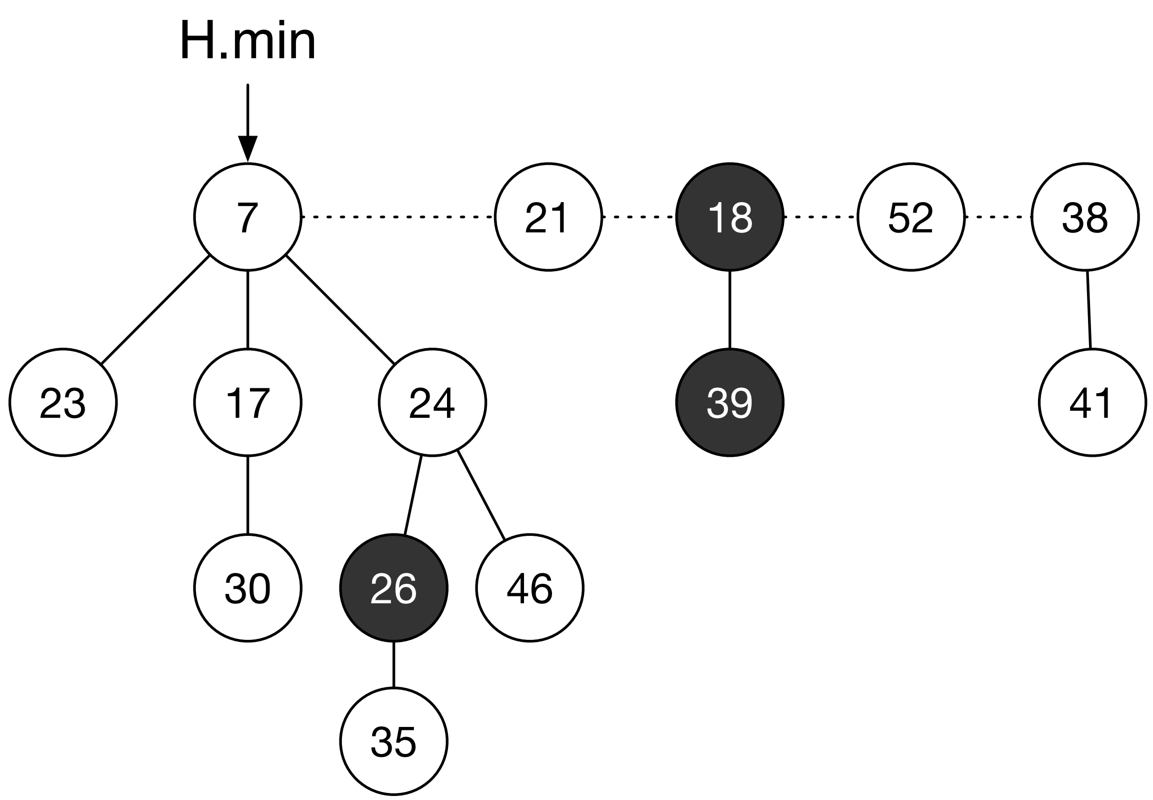
Fibonacci Heaps



A[0]: 21
A[1]: 18
A[2]: NULL
A[3]: 7

And then insert 18.

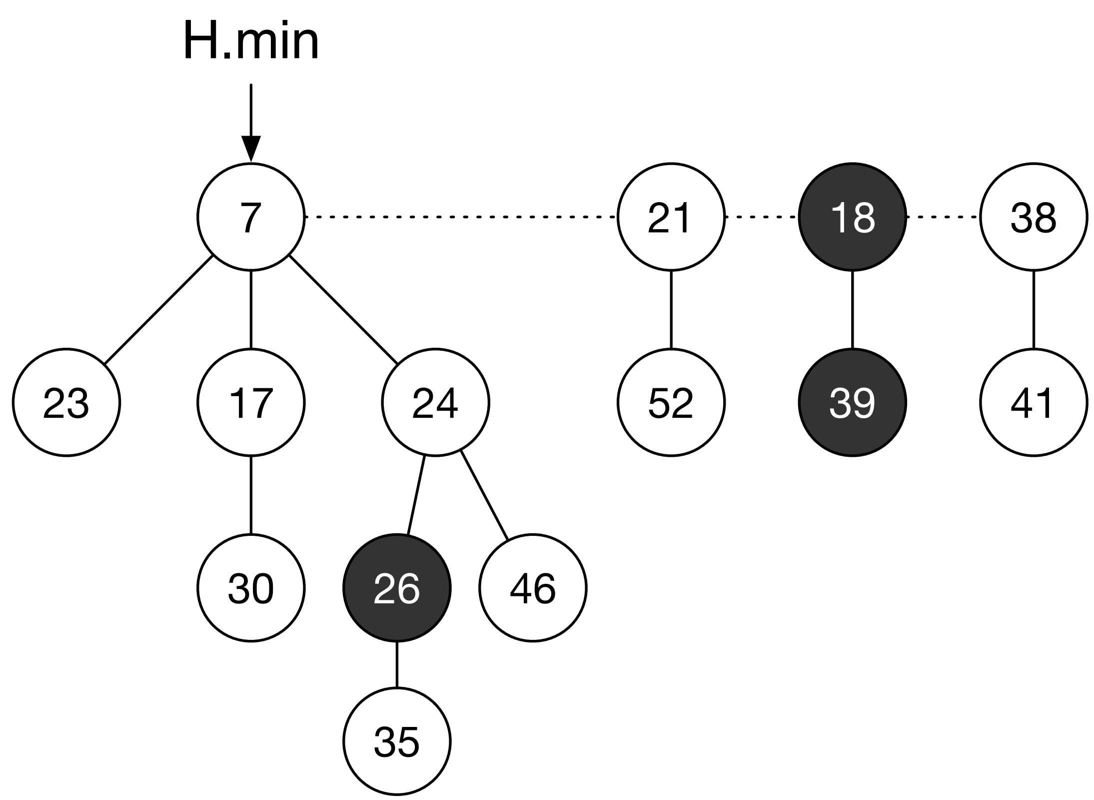
Fibonacci Heaps



A[0]: 21, 52
A[1]: 18
A[2]: NULL
A[3]: 7

Inserting 52 gives us another merge candidate

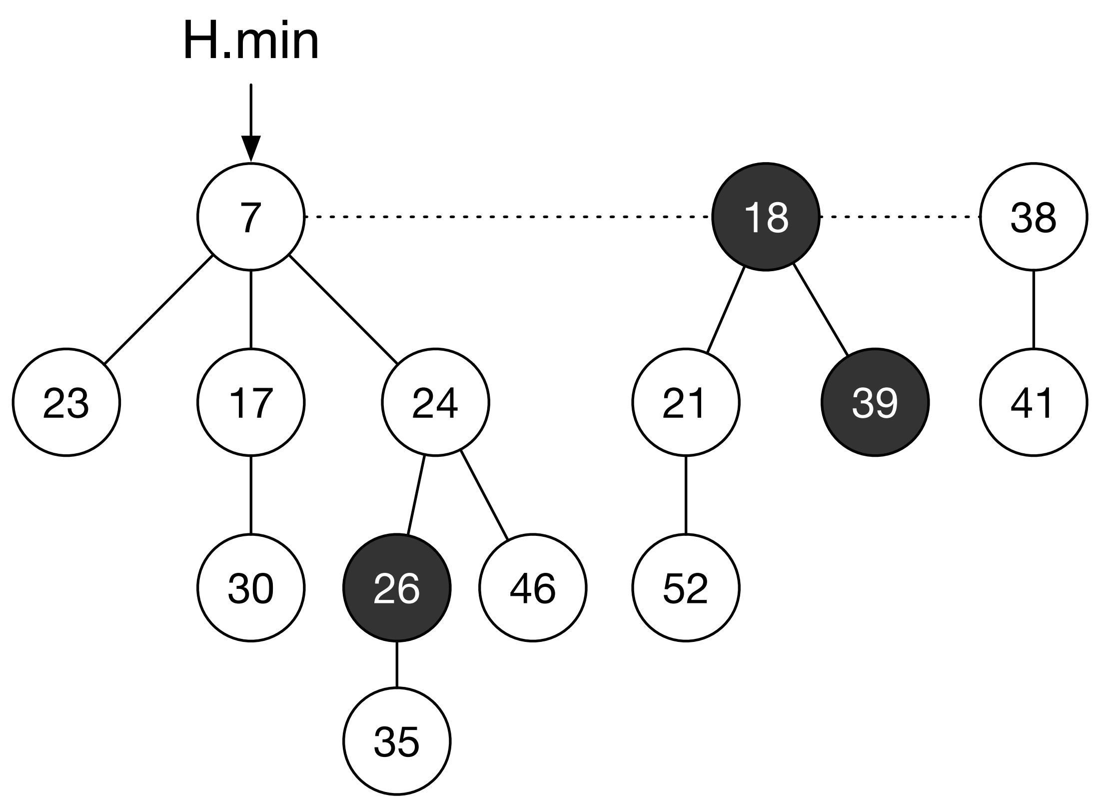
Fibonacci Heaps



A[0]: NULL
A[1]: 18, 21
A[2]: NULL
A[3]: 7

Which leads to another merger

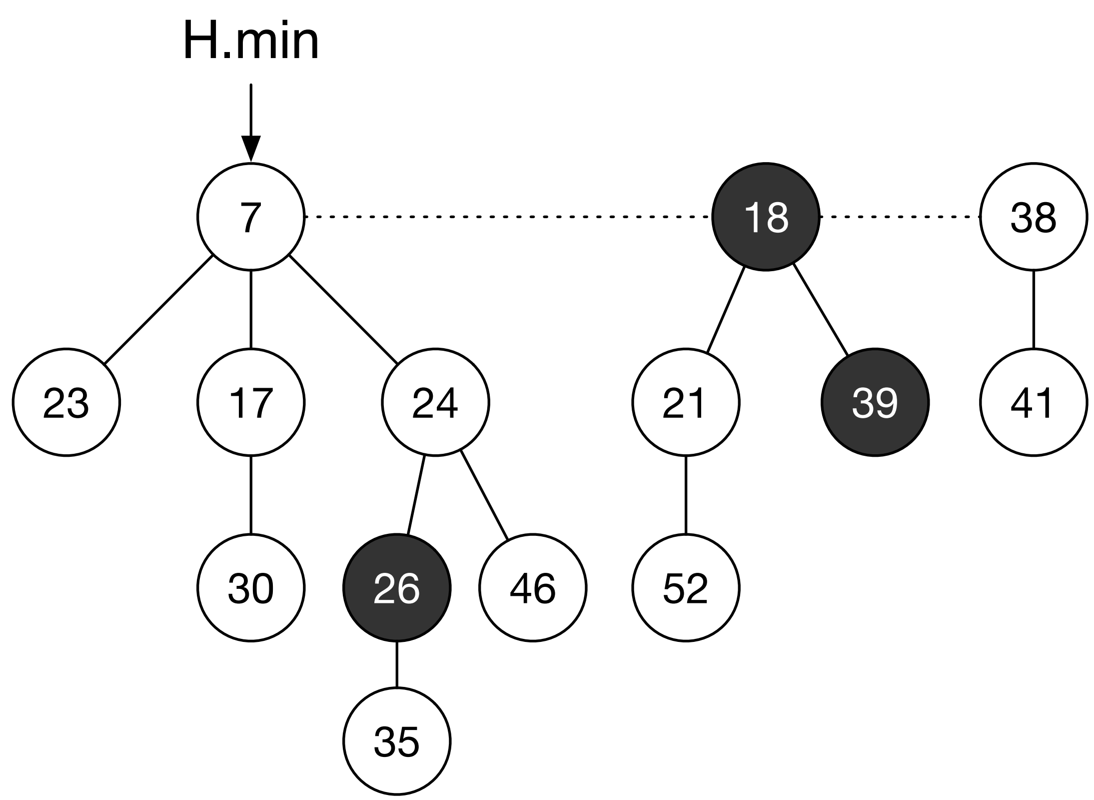
Fibonacci Heaps



A[0]: NULL
A[1]: NULL
A[2]: 18
A[3]: 7

We move on to the next node

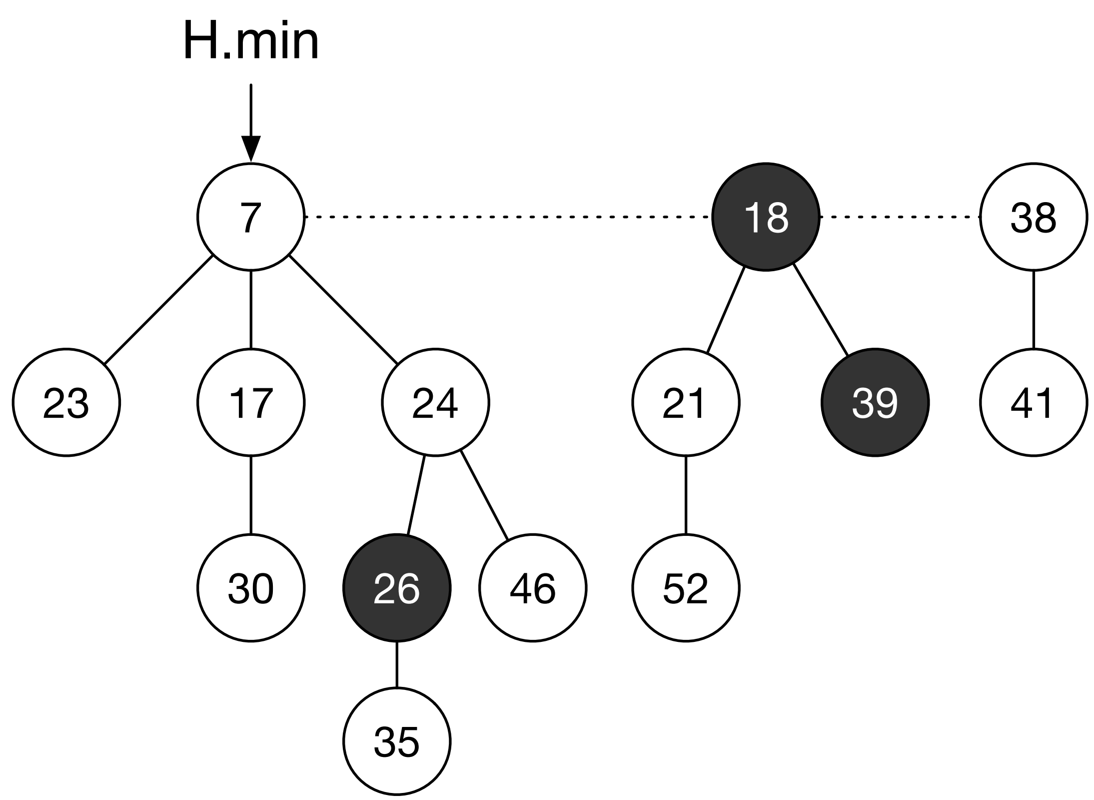
Fibonacci Heaps



A[0]: NULL
A[1]: 38
A[2]: 18
A[3]: 7

We insert it into the array.

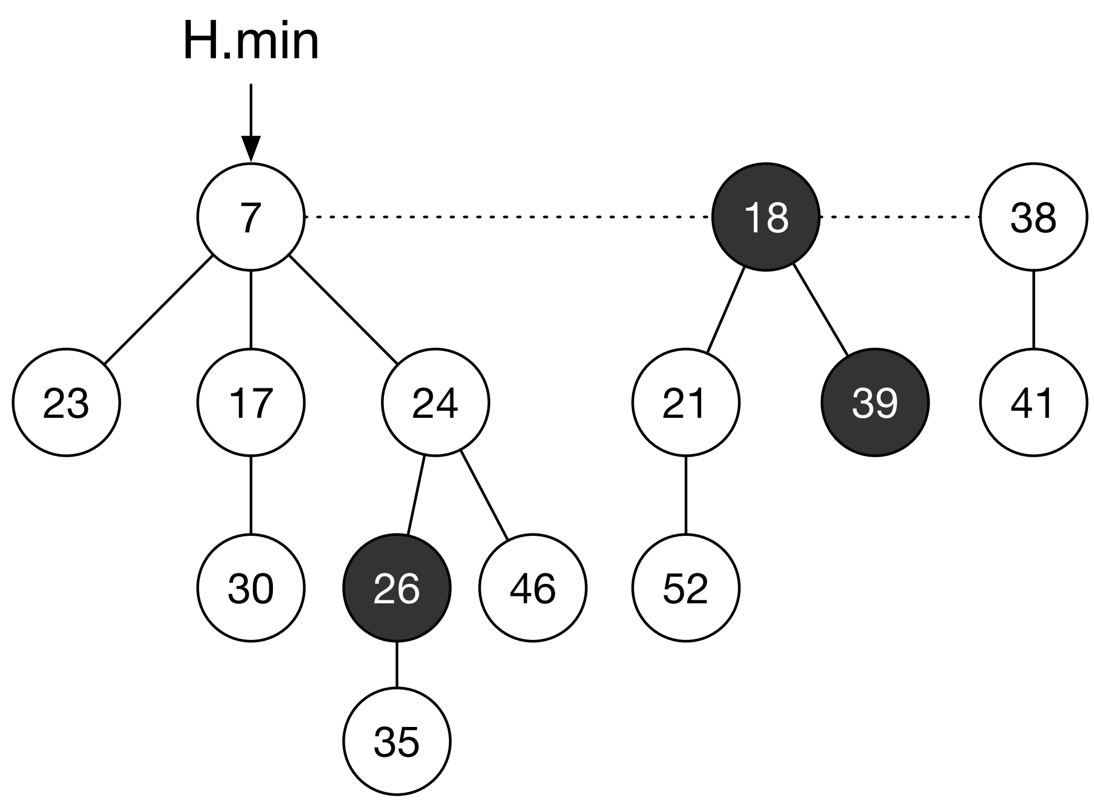
Fibonacci Heaps



A[0]: NULL
A[1]: 38
A[2]: 18
A[3]: 7,7

We then move on to the next node. When we insert, we see that this one is already in A and that we are done merging.

Fibonacci Heaps



A[0]: NULL
A[1]: 38
A[2]: 18
A[3]: 7

We now find the new minimum.

Fibonacci Heaps

- Combining two nodes
  - x.key < y.key

```
def fib_heap_link(H, y, x):
   Assert x.key < y.key
   remove y from root list of H
   y.mark = False
```

Fibonacci Heaps

- Consolidate:

```
def consolidate(H):
    A = [None for i in range(D(H.n))]
    for w in H.root_list:
        while(A[w.degree]):
            degree = A[w.degree]
            y = A[degree]  # candidate for merger
            if w.key > y.key:
                w, y = y, w
            fib_heap_link(H,y,w)
            A[degree] = Null
            degree += 1
        A[w.degree] = w
    reset H.min
```

Fibonacci Heaps

- Costs:
  - Potential:
    - Potential before is t(H)+2m(H)
    - Potential after is D(n)+1+2m(H)
      - because A has D(n) entries
      - because nobody gets marked
  - Work done:
    - t(H) for going through the root list
    - ≤ D(n) for inserting the children of minimum
  - Amortized costs:
  - $\leq O(D(n) + t(H)) + \big(D(n) + 1 + 2m(H) - t(H) - 2m(H)\big) = O(D(n))$

Fibonacci Heaps

- Decrease a key
    - Find parent p
        - if p and x.key < p.key:
            - CUT(H,x,y)
            - CASCADING-CUT(H,y)
    - If necessary, adjust H.min

Fibonacci Heaps

- Cut(H, x, y)

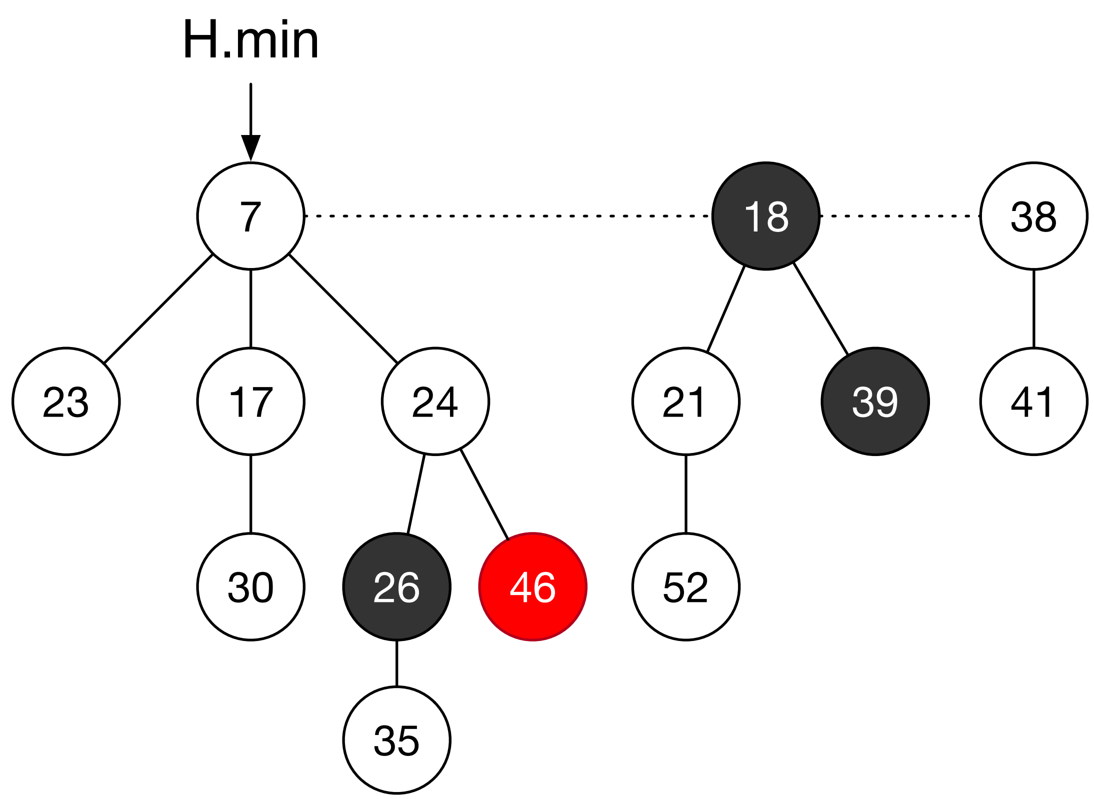
```
def cut(H, x, y):
   remove x from child list of y
   y.degree -= 1
   add x to root list of H
   x.p = Null
   x.mark = False
```

Fibonacci Heaps

- Cascading Cut

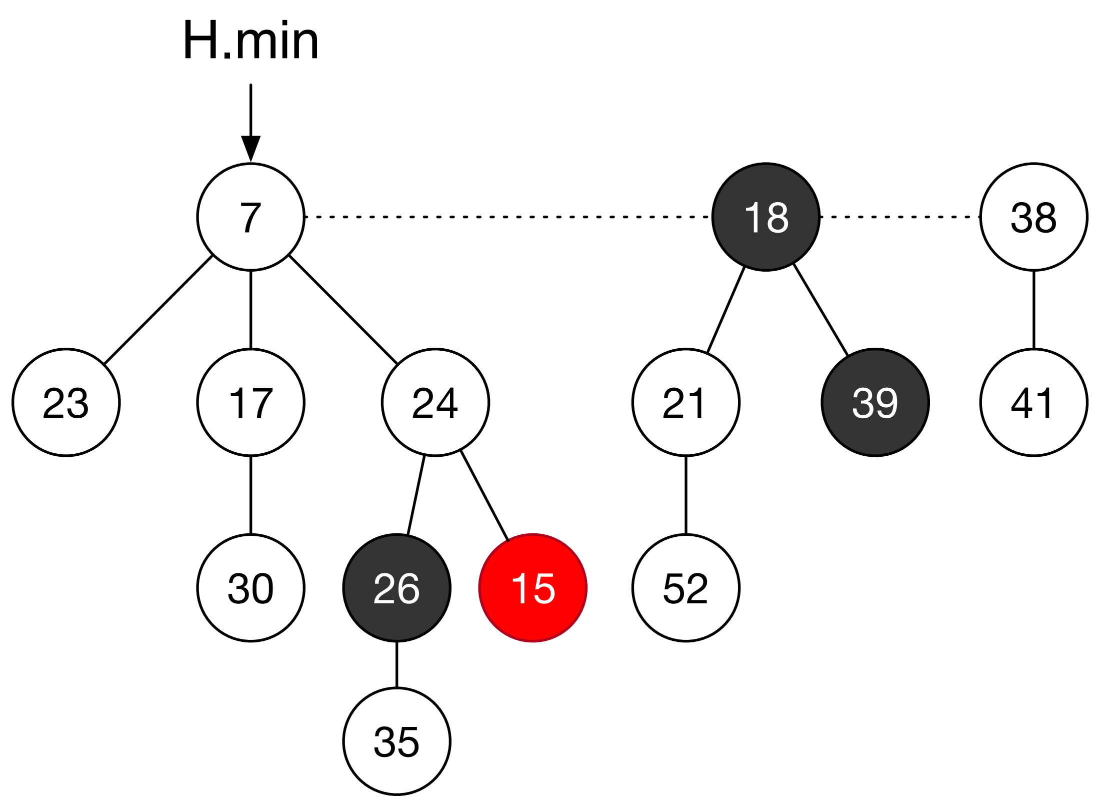
```
def cascading_cut(H,y):
    z = y.p # parent of y
    if z:
        if y.mark == False:
            y.mark = True
        else:
            cut(H, y, z)
            cascading_cut(H,z)
```

Fibonacci Heaps

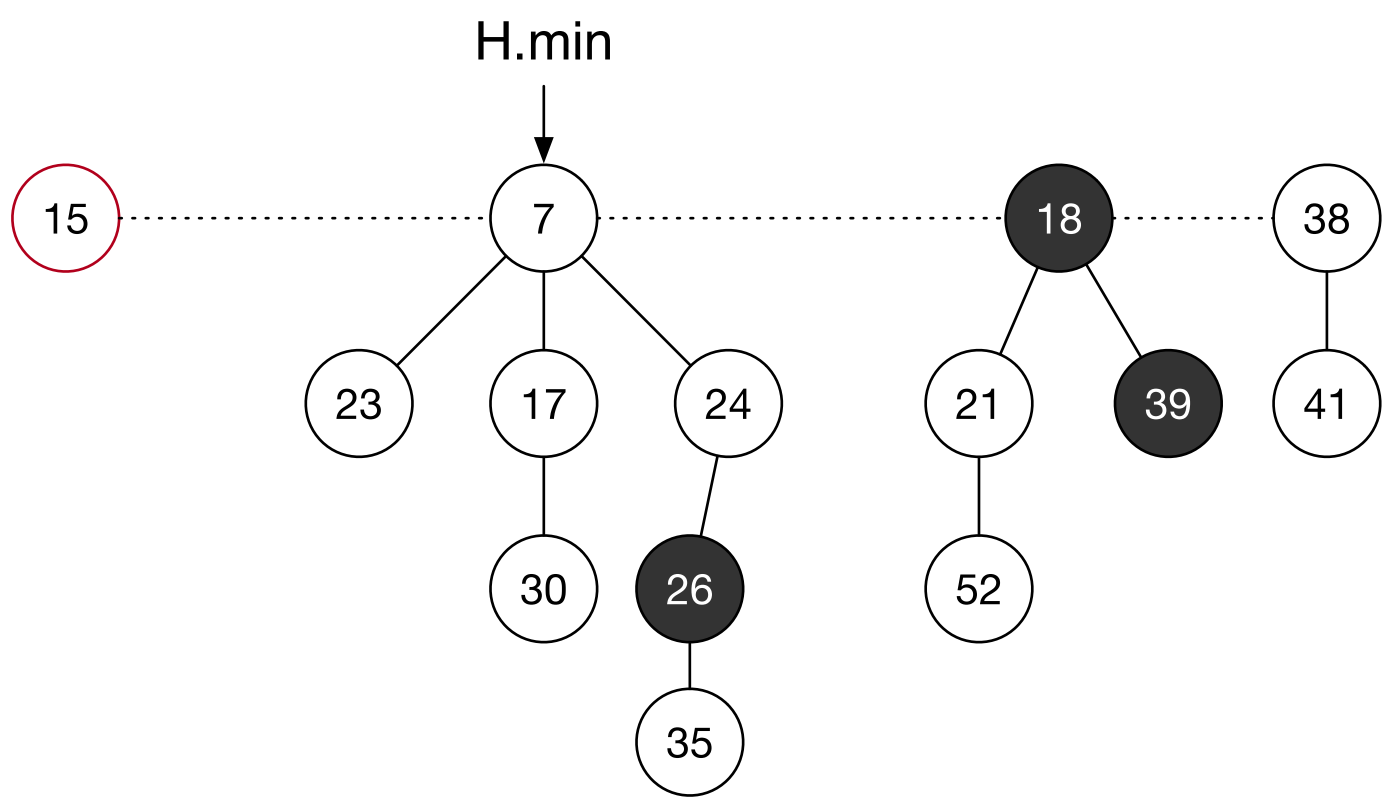


Decrease key to 15

Fibonacci Heaps

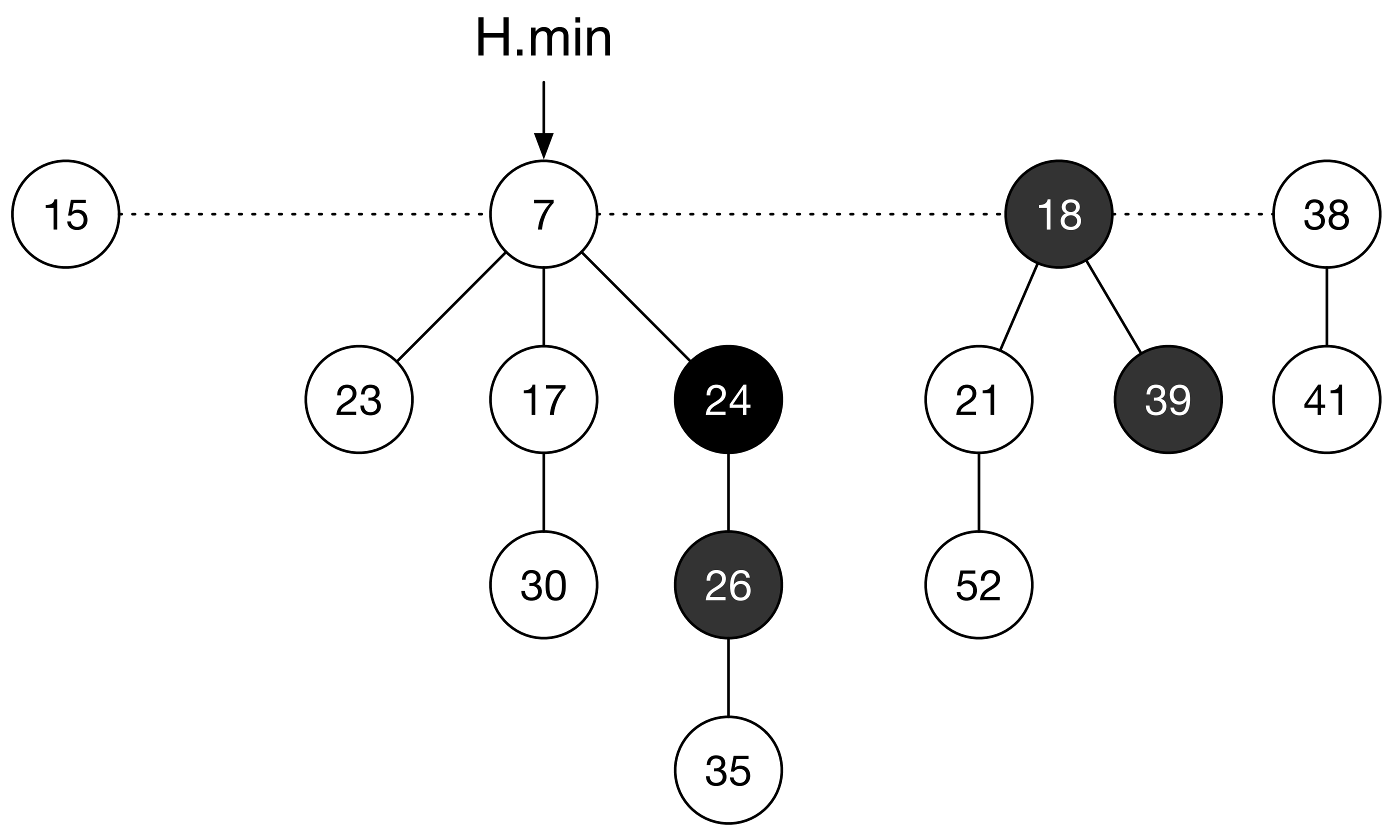


Fibonacci Heaps



Remove node (and any descendants)

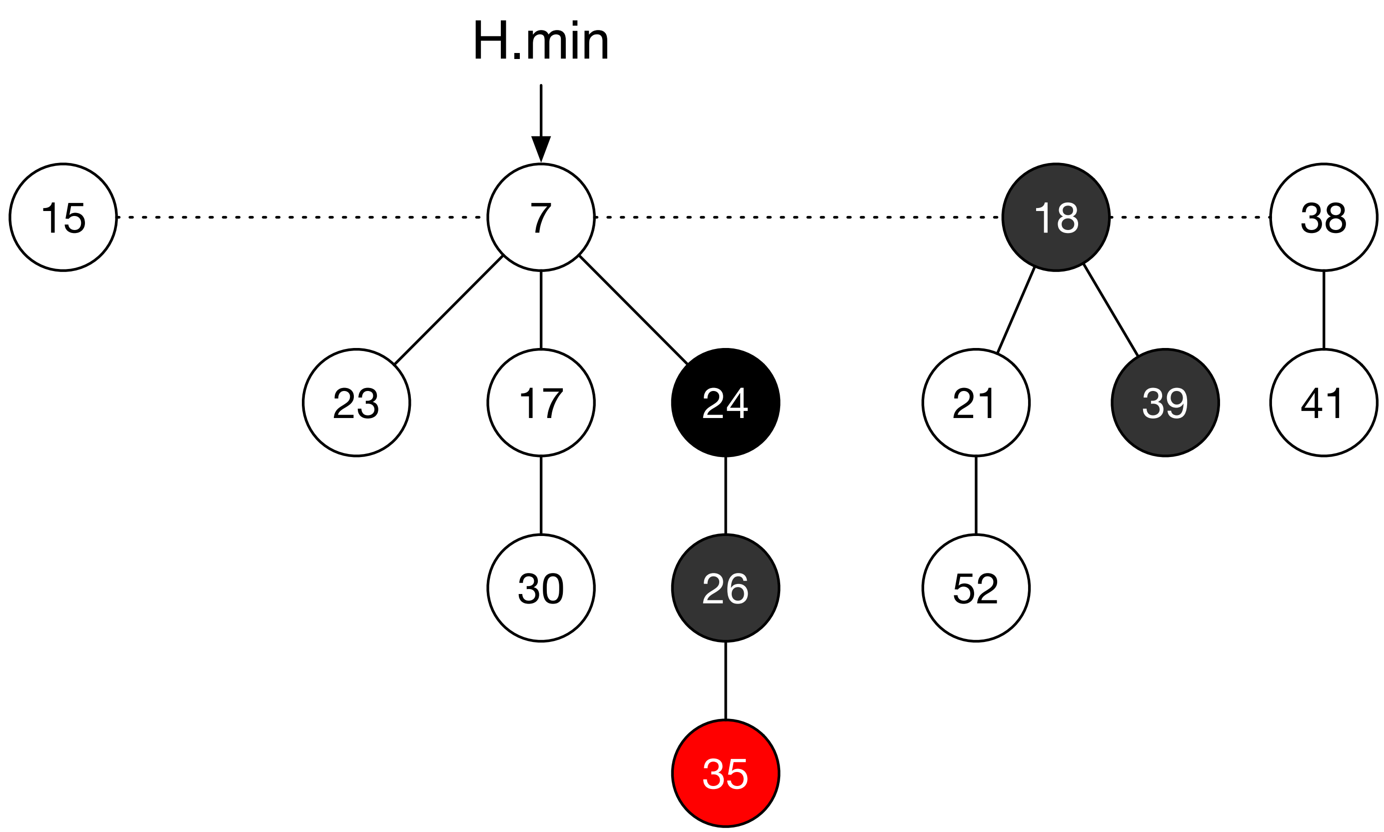
Fibonacci Heaps



Mark parent!
Done!

Fibonacci Heaps

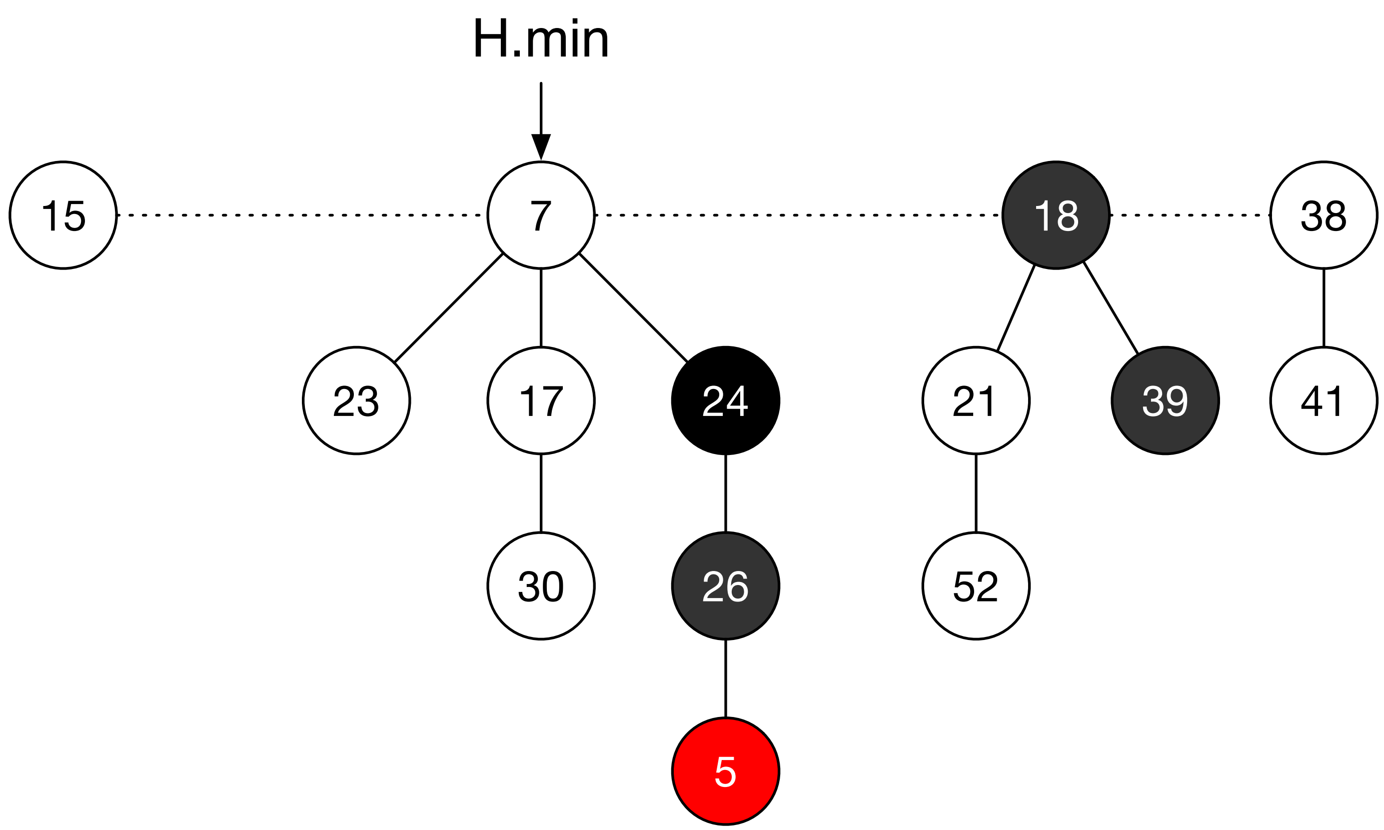
Second operation



Change key from 35 to 5

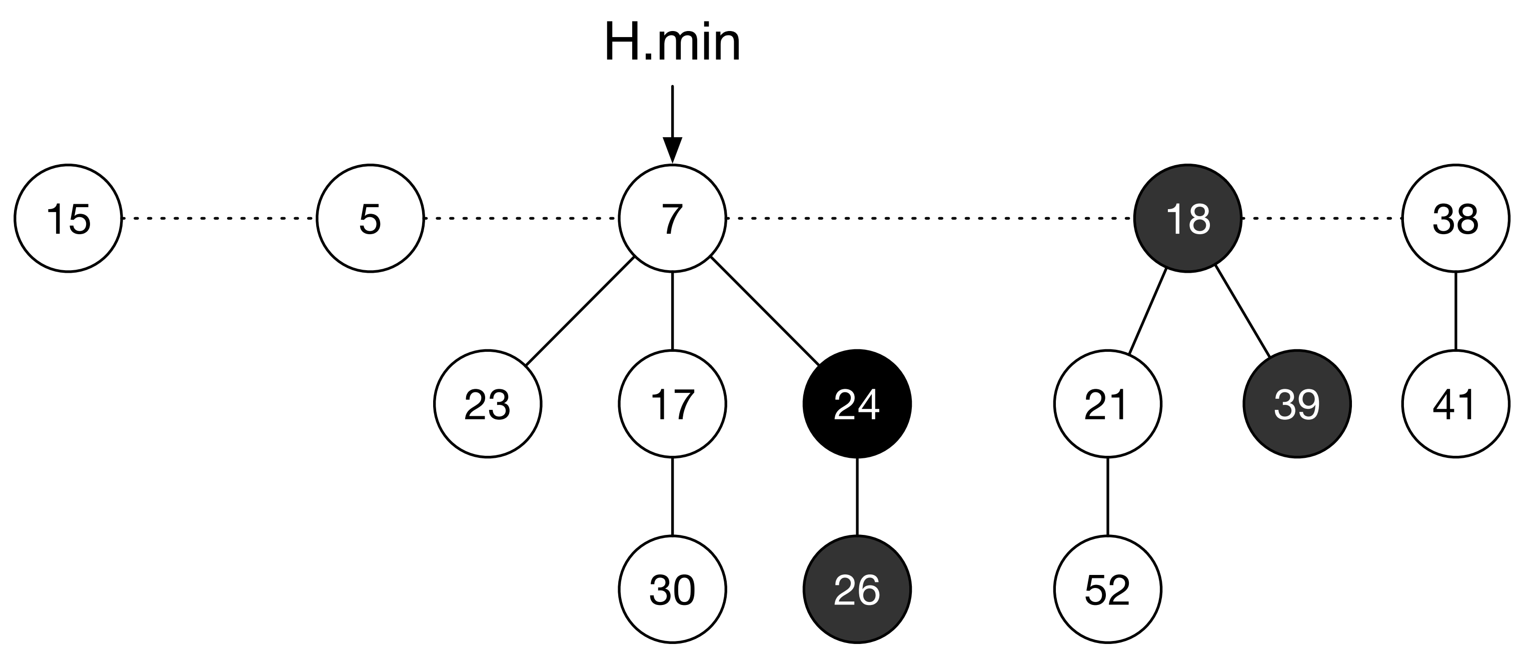
Fibonacci Heaps

Second operation



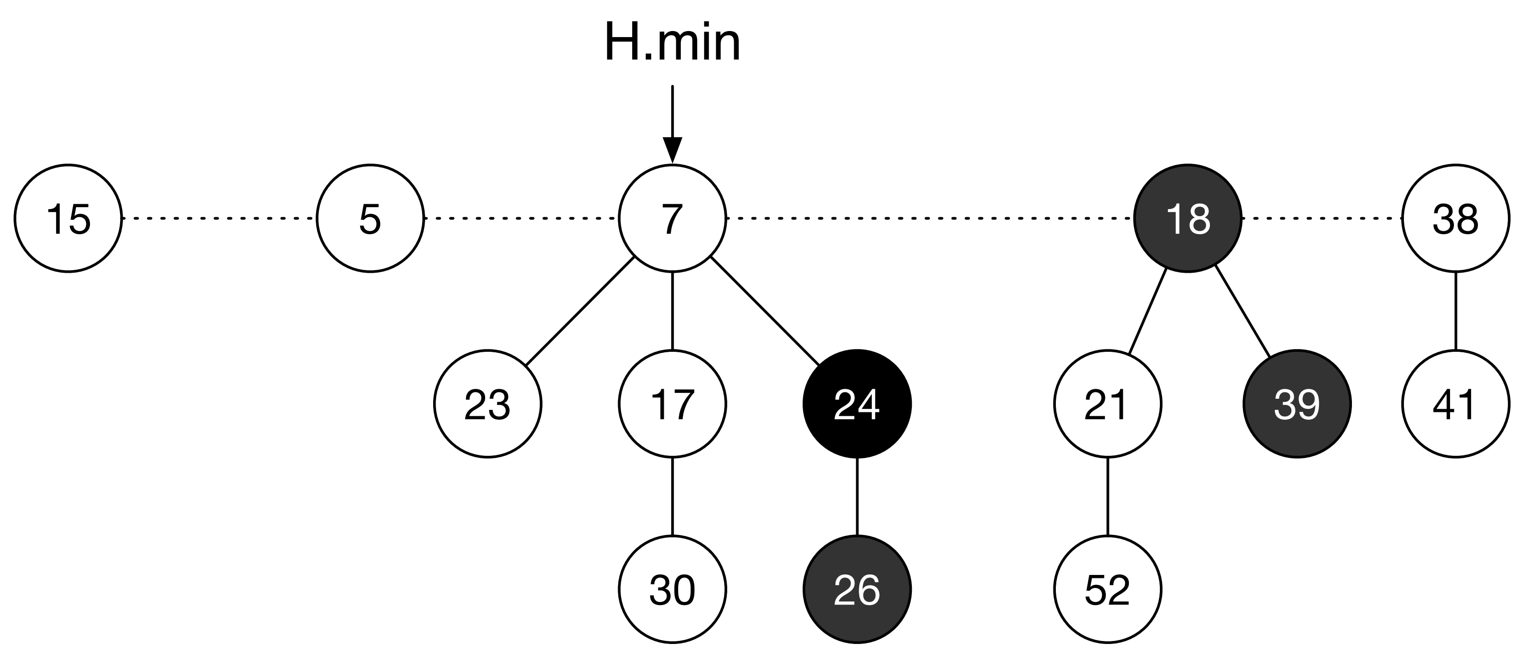
Change key from 35 to 5

Fibonacci Heaps



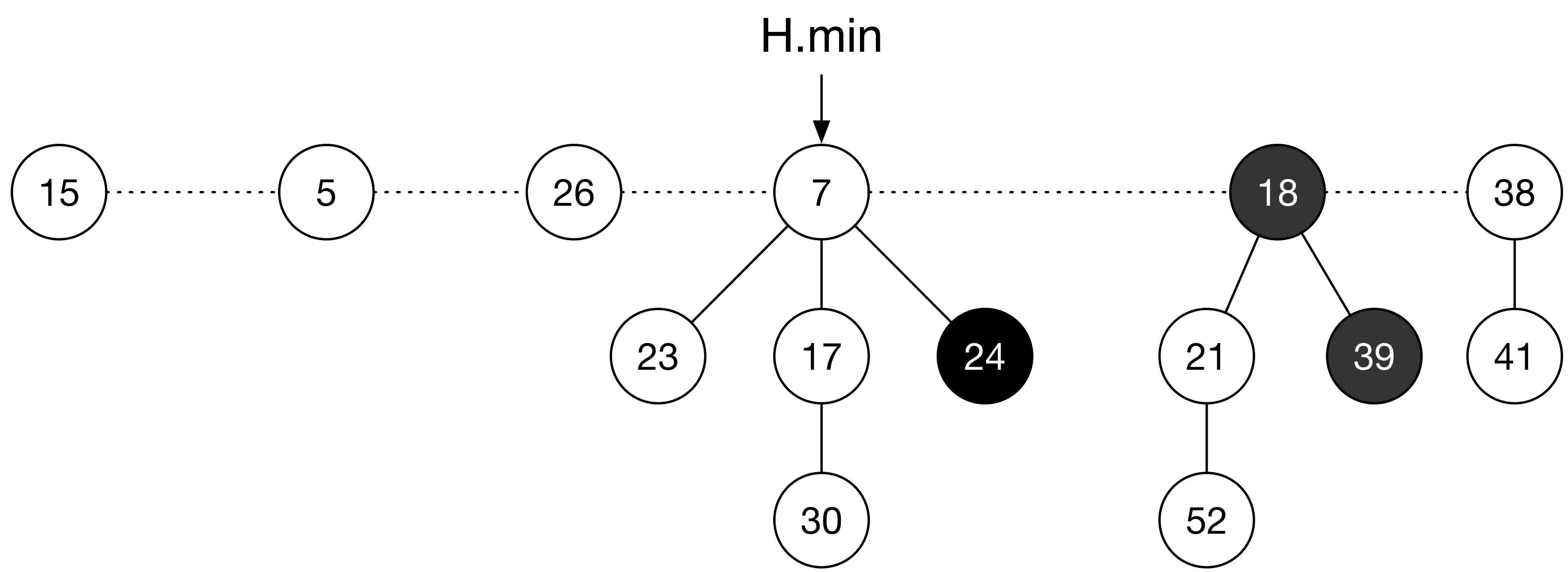
Cut node.

Fibonacci Heaps



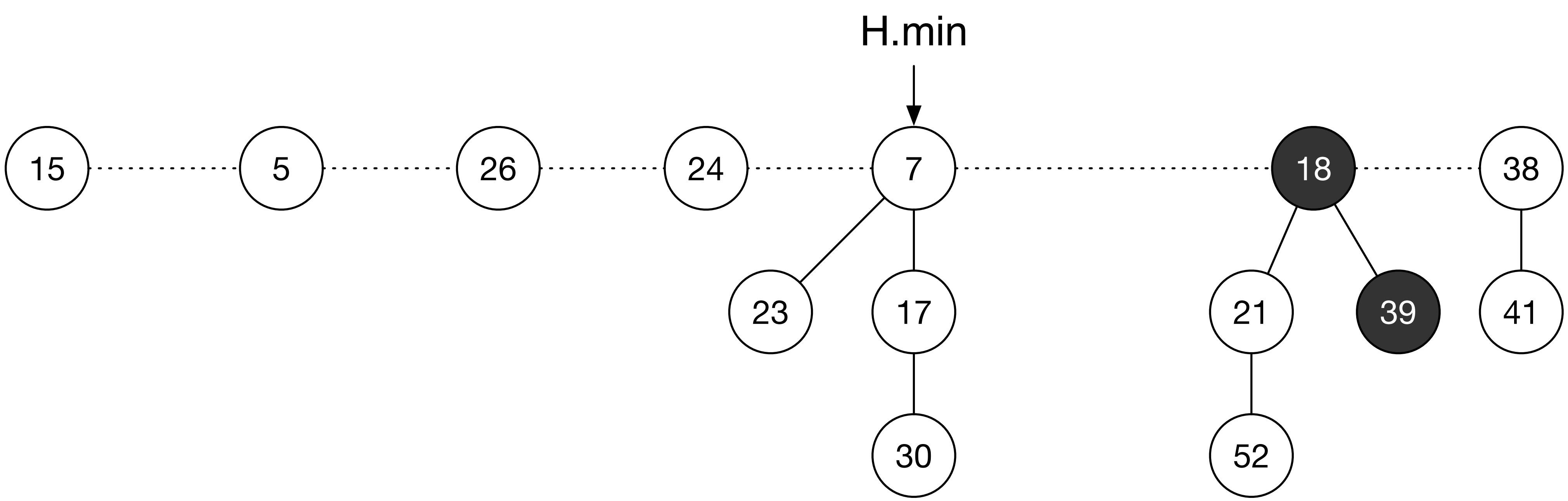
Mark parent.
But parent is already marked, so:
Cascading Cut!

Fibonacci Heaps



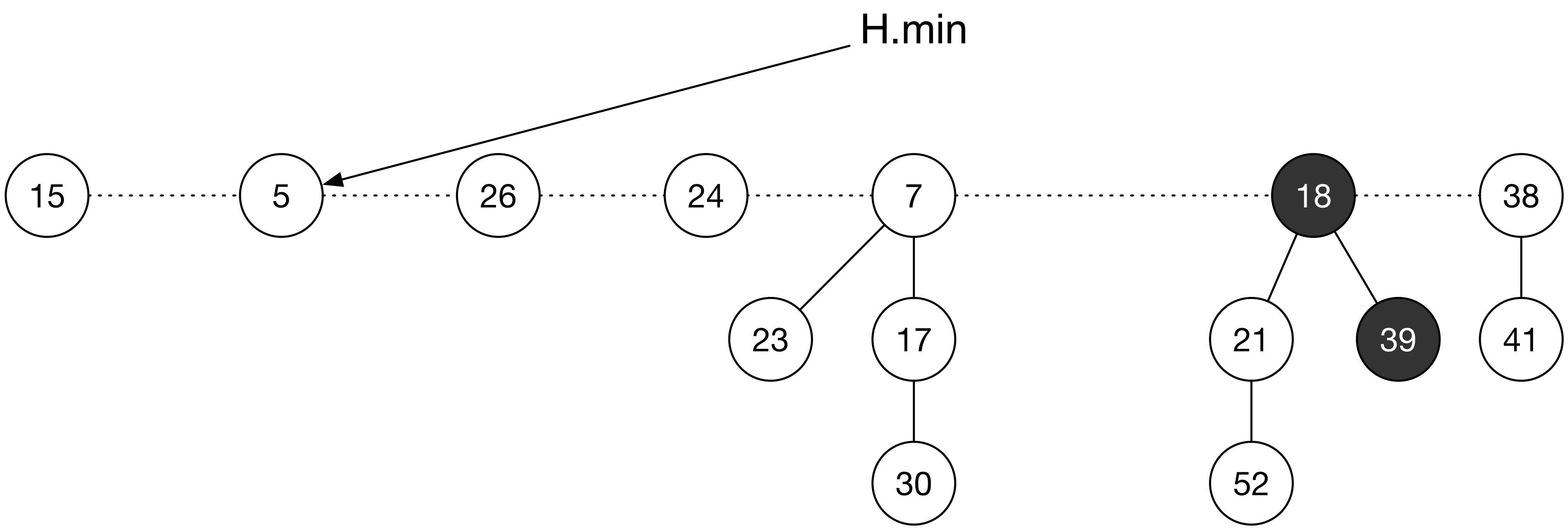
26 is removed.
Look at 24: It is also marked

Fibonacci Heaps



Cut 24, unmarking it.
Cut stops here, parent is unmarked.

Fibonacci Heaps



Reset minimum

Fibonacci Heaps

- Change in potential:
  - Cut creates a new tree and potentially clears a mark
  - Each Cascading-Cut cuts a marked node and clears the mark bit (with exception of the last one)
- H now has c-1 trees produced by cascading cuts and the tree at x: t(H)+c trees
- At most m(H)-c+2 marked nodes
- $\leq t(H) + c + 2(m(H) - c + 2) - t(H) - 2m(H) = 4 - c$

Fibonacci Heaps

- Amortized cost of decrease key:
  - $O(c) + 4 - c = O(1)$

Fibonacci Heaps

- Deleting a node
  -
    ```
    def delete(H,x):
        decrease_key(H,x,-infty)
        extract_min(H)
    ```

Fibonacci Heaps

- Left to investigate:
  - Upper bound *D*(*n*) on the degrees is $O(\log(n))$
- Lemma: $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$

Fibonacci Heaps

- Lemma: $F_{k+2} \geq \phi^k$ with $\phi = \dfrac{1+\sqrt{5}}{2}$

Fibonacci Heaps

- Lemma: Let *x* be any node in a Fibonacci heap with x.degree = *k*. Let $y_1, y_2, \ldots, y_k$ be the children of *x* in the order in which they were linked to *x* (by consolidate). Then:

  y_1 . degree $\geq 0, \quad y_i$. degree $\geq i - 2$ for all $i = 2,3,\ldots,k$

Fibonacci Heaps

- Proof:
  - Clearly, y_1 . degree ≥ 0
  - Assume a general $i \geq 2$.
    - When CONSOLIDATE links y_i to x, then all of $y_1, y_2, \ldots$ y_{i-1} was linked to x
    - This means x . degree $\geq i$
    - Because y_i is linked to x only if x . degree $= y_i$. degree
      - y_i . degree $\geq i - 1$

Fibonacci Heaps

- But in the mean-time, the degree of y_i might have decreased
- But Cascading-Cut would cut y_i from x if y_i has lost more than two children
- Therefore y_i . degree $\geq i - 2$

Fibonacci Heaps

- Lemma: Let x be any node in a Fibonacci heap and let $k = x$. degree. Then size$(x) \geq F_{k+2} \geq \Phi^k$.
    - Let s_k denote the minimum possible size of any node of degree k in a Fibonacci heap.
    - $s_0 = 1, \qquad s_1 = 2$
    - Adding children does not decrease the node's size, the value of s_k increases monotonically with *k.*

Fibonacci Heaps

- Notice $s_k \leq \text{size}(x)$, so giving a lower bound on s_k is sufficient
- Take a node z with
  - $z\,.\,\text{degree} = k \qquad \text{size}(z) = s_k$
- Let $y_1, y_2, \ldots, y_k$ denote the children of z in the order in which they were linked to z
- To bound s_k, we have one for z and one for y_1 and the rest

Fibonacci Heaps

- This gives
  - size$(x) \geq s_k$
  - $\geq 2 + \sum_{i=2}^{k} s_{y_i.\text{degree}}$
  - $\geq 2 + \sum_{i=2}^{k} s_{i-2}$ because y_i . degree $\geq i - 2$ by Lemma

    and monotonicity of s_k

Fibonacci Heaps

- We prove by induction that $s_i \geq F_{i+2}$
- Induction step:
- $s_k \geq 2 + \sum_{i=2}^{k} s_{i-2}$
- $\geq 2 + \sum_{i=2}^{k} F_i$
- $= 1 + \sum_{i=0}^{k} F_i$
- $=F_{k+2} \geq \Phi^k$

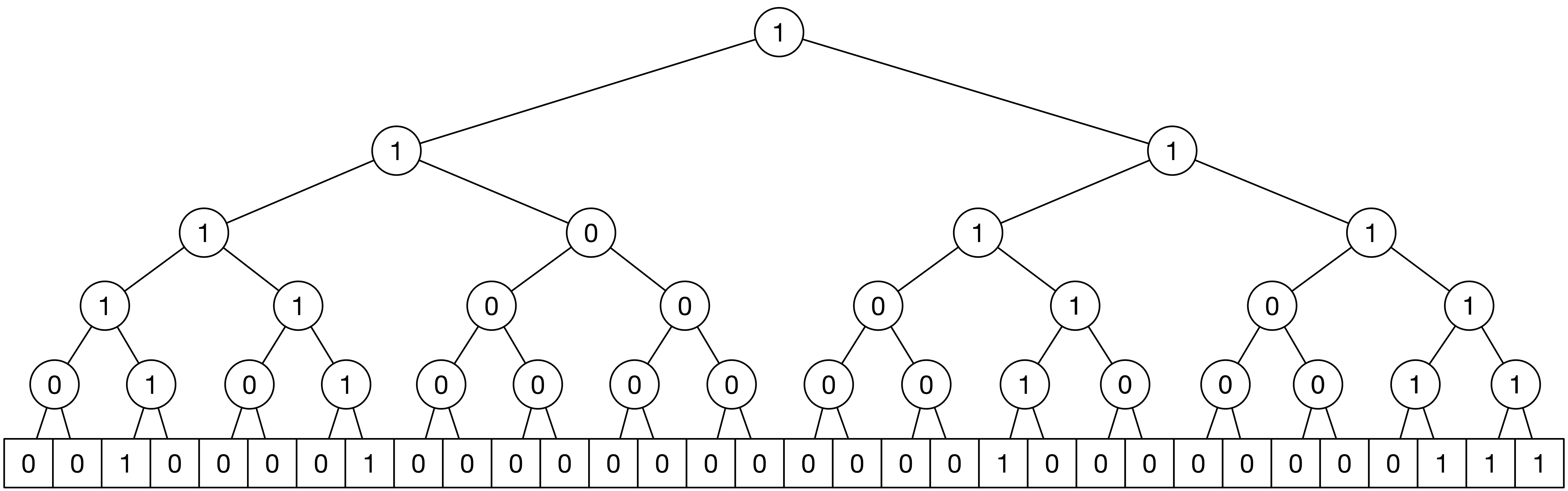
Fibonacci Heaps

- Let x be the node with maximum degree into an n-node Fibonacci Heap
- Degree of x is k
- By lemma, $n \geq \text{size}(x) \geq \Phi^k$
- Thus: $\log_\Phi(n) \geq k$
- Thus: maximum degree is $O(\log n)$

van Emde Boas Trees

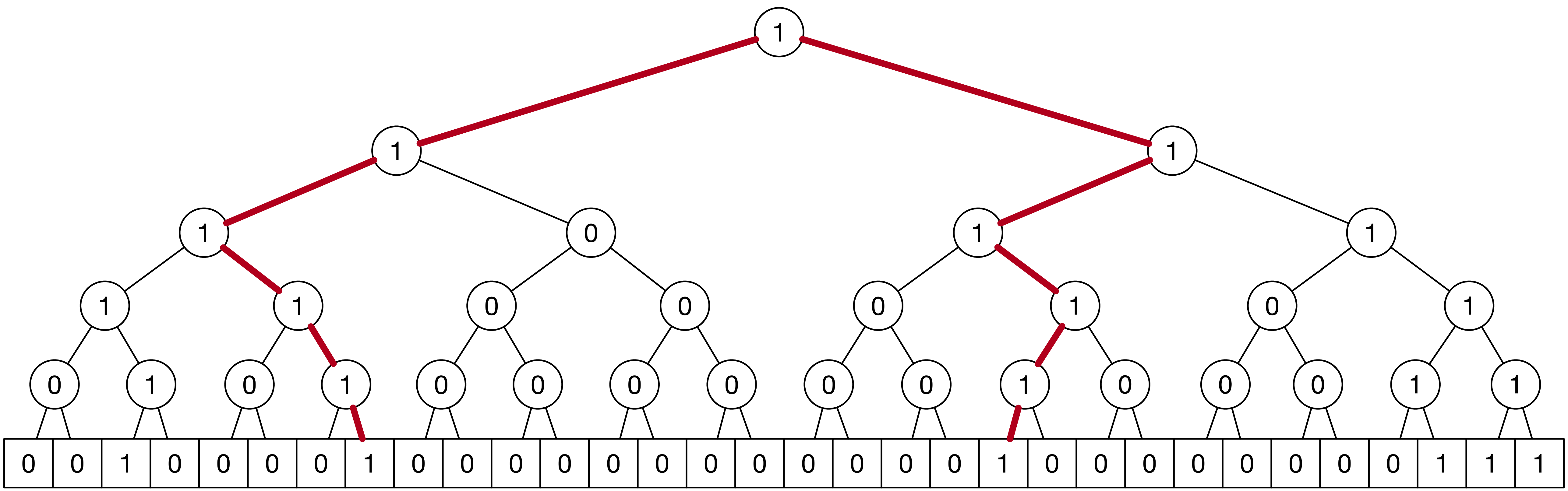
- Operations of priority queues have at least one logarithmic operation
- If everything would be less, then we could sort in time $n \cdot \log n$
- But we can sort an array of integers in range 1 … *n* in time O(*n*)

van Emde Boas Trees



Binary tree on top of a bitvector

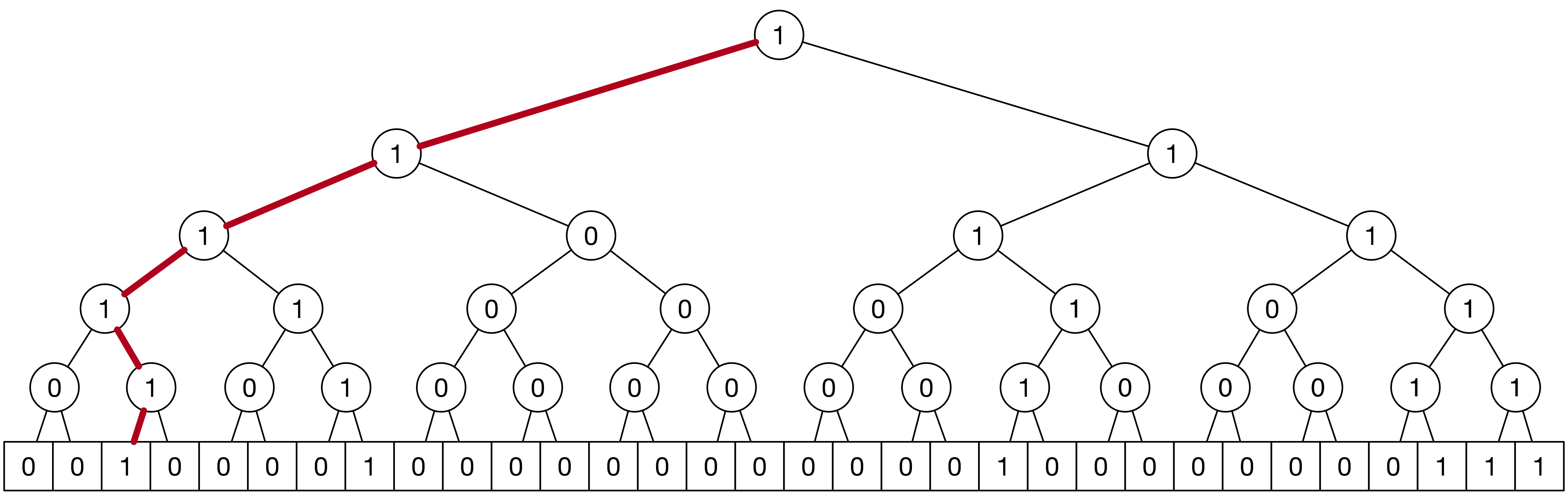
van Emde Boas Trees



Finding the NextIndex / PreviousIndex
is logarithmic

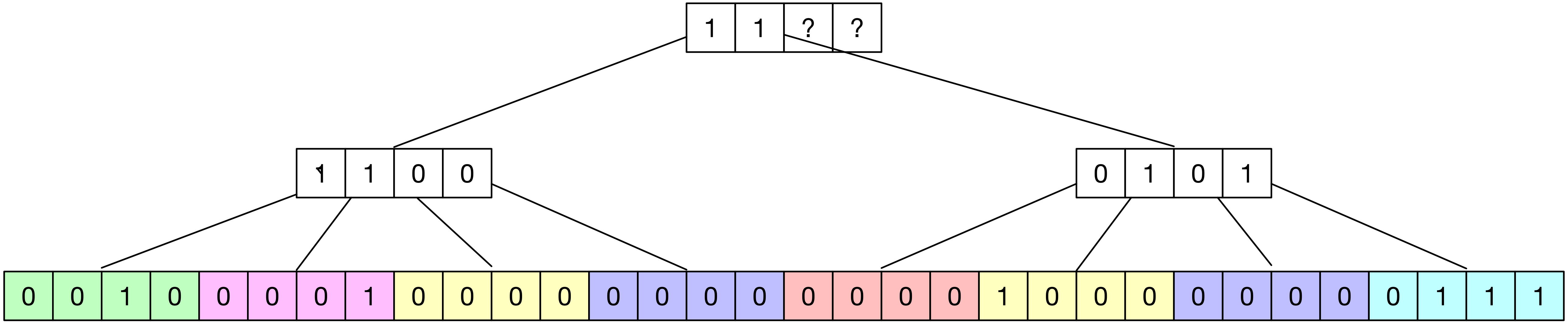
van Emde Boas Trees

- Finding the minimum value in logarithmic time:
  - Start at root
  - Head down taking the leftmost one



van Emde Boas Trees

- We can combine nodes into a tree with higher fan-out

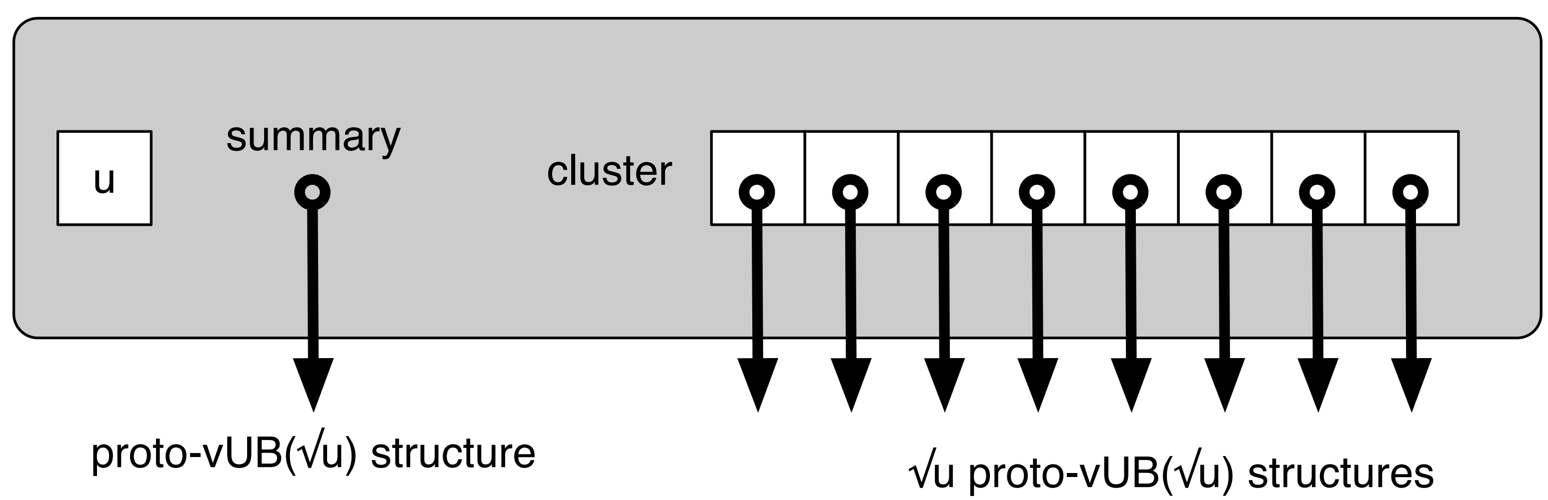


van Emde Boas Trees

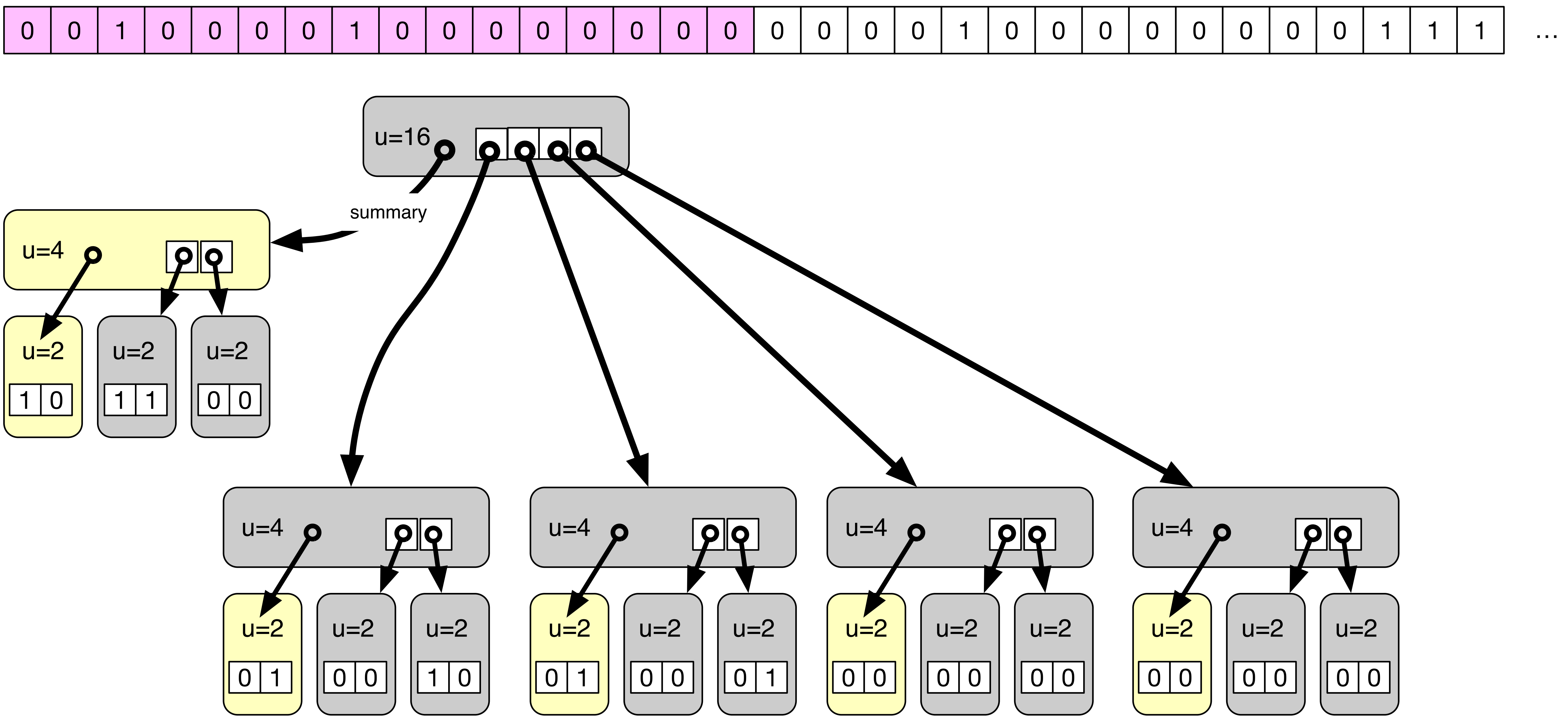
- Assume that the number u of elements in the universe is $u = 2^{2^k}$
- To build a recursive structure:
  - Recurrence $T(u) = T(\sqrt{u}) + O(1)$
    - Setting $m = \log_2(u)$, $S(m) = T(2^m)$
    - Recurrence is $S(m) = S(m/2) + O(1)$
    - MT $\implies S(m) = O(\log_2(m))$
    - $T(u) = T(2^m) = S(m) = O(\log_2(m)) = O(\log_2(\log_2(u)))$

van Emde Boas Trees

- Proto-vEM-structures for u
  - $u = 2$: array of two bits
  - $u = 2^{2^k}$ with $k \geq 1$:
    - summary pointer towards parent
    - cluster towards children



van Emde Boas Trees



van Emde Boas Trees

- Operations on proto-vEB-Trees
  - For a given $x \in U$:
    - $\text{high}(x) = \lfloor \frac{x}{\sqrt{u}} \rfloor$ number of cluster
    - $\text{low}(x) = x \pmod{\sqrt{u}}$ position of x in cluster
    - $\text{index}(x, y) = x\sqrt{u} + y$ rebuilds index from number of cluster and position in cluster

van Emde Boas Trees

- Operations on proto-vEB-Trees
  - Membership

```
def pvEB_member(V,x):
    if V.u == 2:
        return V.A[x]
    else:
        return pvEB_member(V.cluster[high(x)].low(x)
```

  - Runtime has recurrence $T(u) = T(\sqrt{u}) + O(1)$

van Emde Boas Trees

- Operations on proto-vEB-Trees
    -

van Emde Boas Trees

- Operations on proto-vEB-Trees

van Emde Boas Trees

- Operations on proto-vEB-Trees

van Emde Boas Trees

- Operations on proto-vEB-Trees

van Emde Boas Trees

- Operations on proto-vEB-Trees

van Emde Boas Trees

- Operations on proto-vEB-Trees

van Emde Boas Trees

van Emde Boas Trees

van Emde Boas Trees

van Emde Boas Trees

van Emde Boas Trees

van Emde Boas Trees

van Emde Boas Trees

van Emde Boas Trees

van Emde Boas Trees

van Emde Boas Trees

van Emde Boas Trees