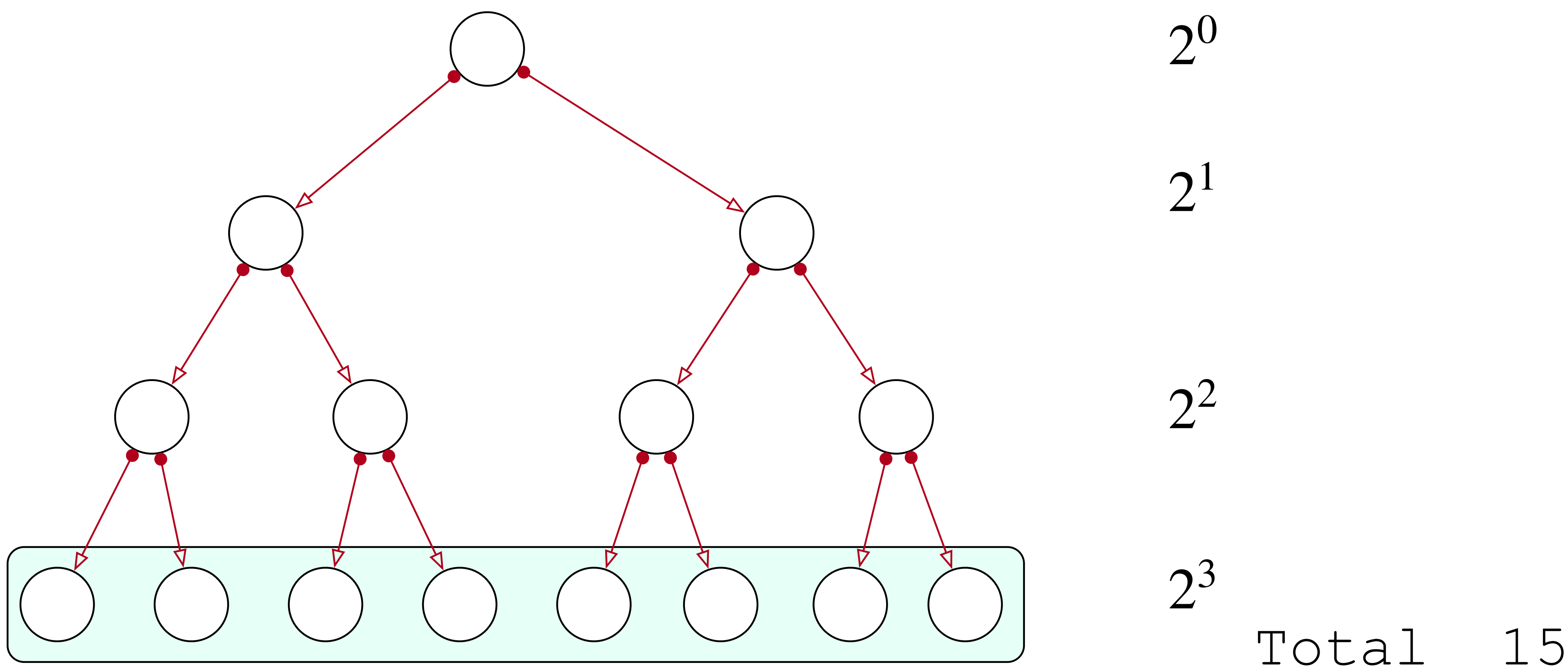
Binary Trees II

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
      - $2^{d-1} \leq m + 1 < 2^d$
      - $d - 1 \leq \log_2(m+1) < d$
      - $d - 1 = \lfloor \log_2(m+1) \rfloor$

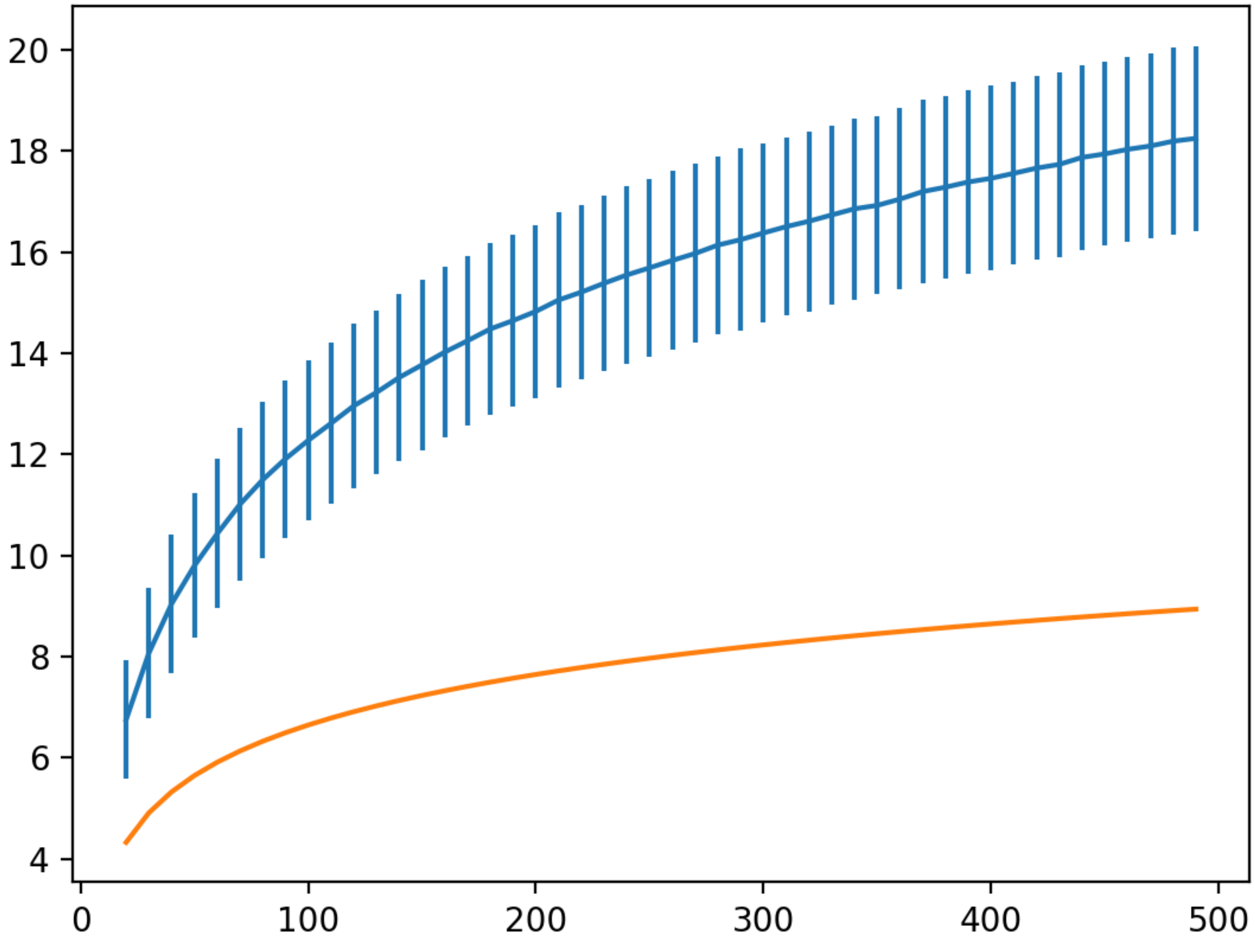
Behavior of Trees

- 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

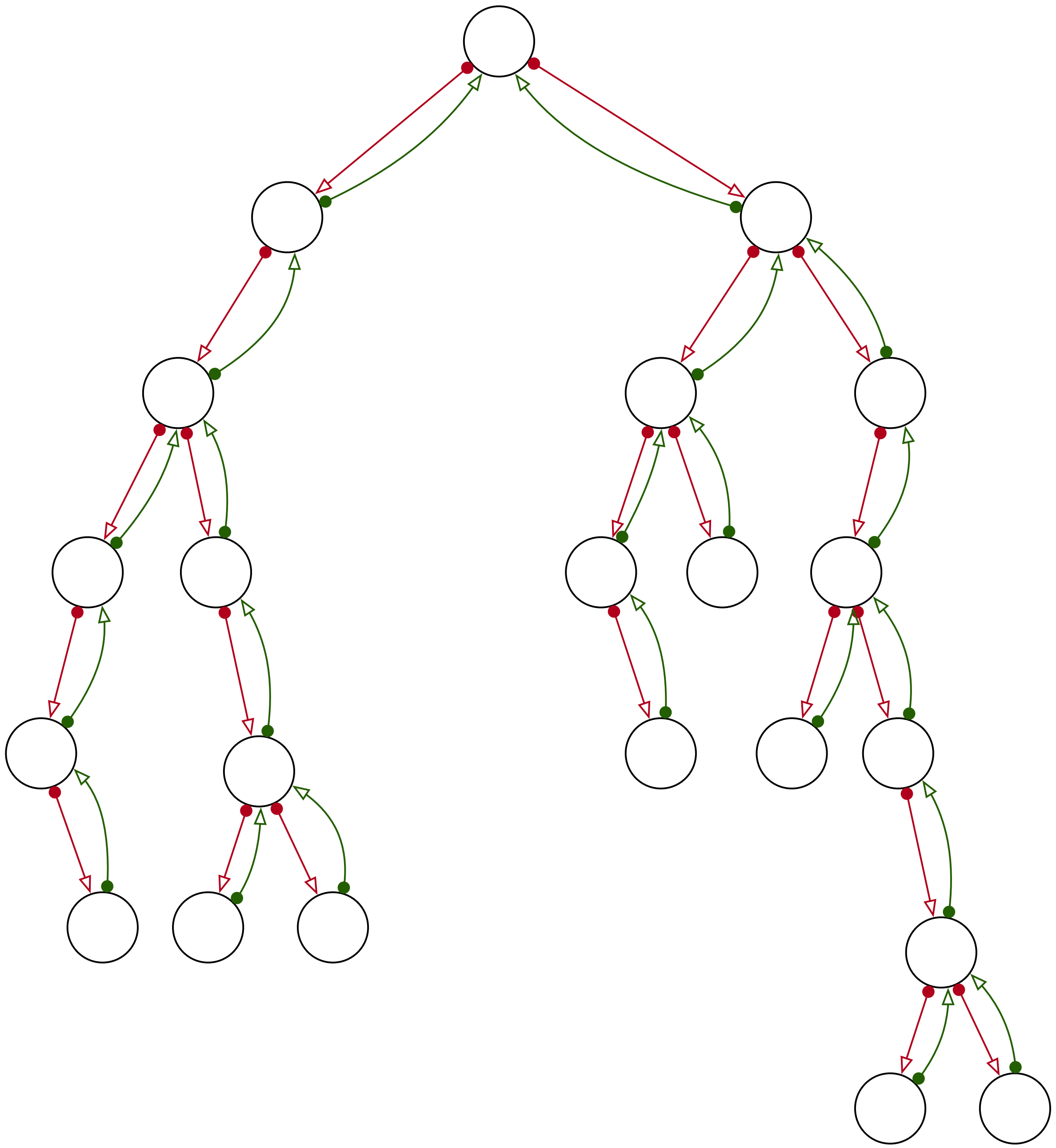
- On average
  - Trees have more than twice necessary depth
- But on average
  - Behavior is still logarithmic
- Theory: Height of a random binary search tree for a random permutation of n elements is
  - $\alpha \log_e(n)$ with $\alpha \approx 4.31107$
  - $= 2.98821 \log_2(n)$
    - Robson 1979 / Devroye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

- Deletion
    - W

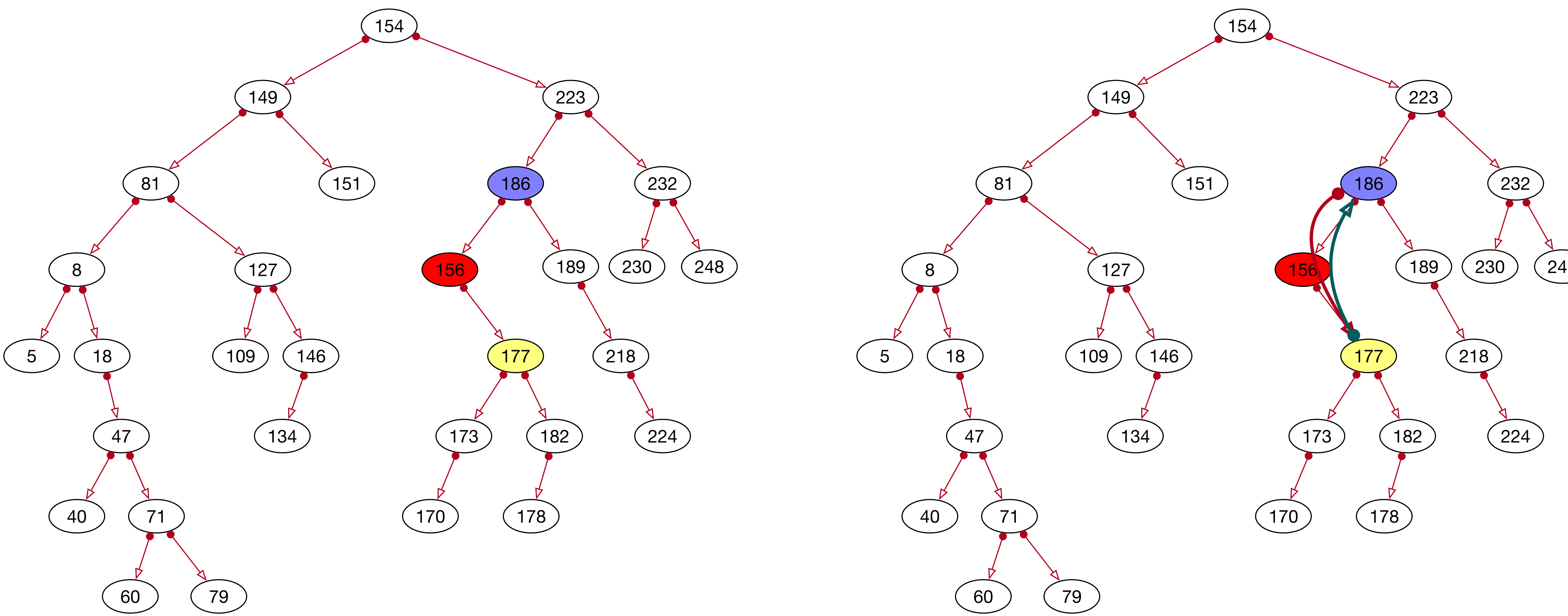
```
def remove(self, value):
        if not self.root:
            return False
        current = self.root
        while True:
            if not current:
                return False
            if value == current.value:
                break
            if value < current.value:
                current = 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
```

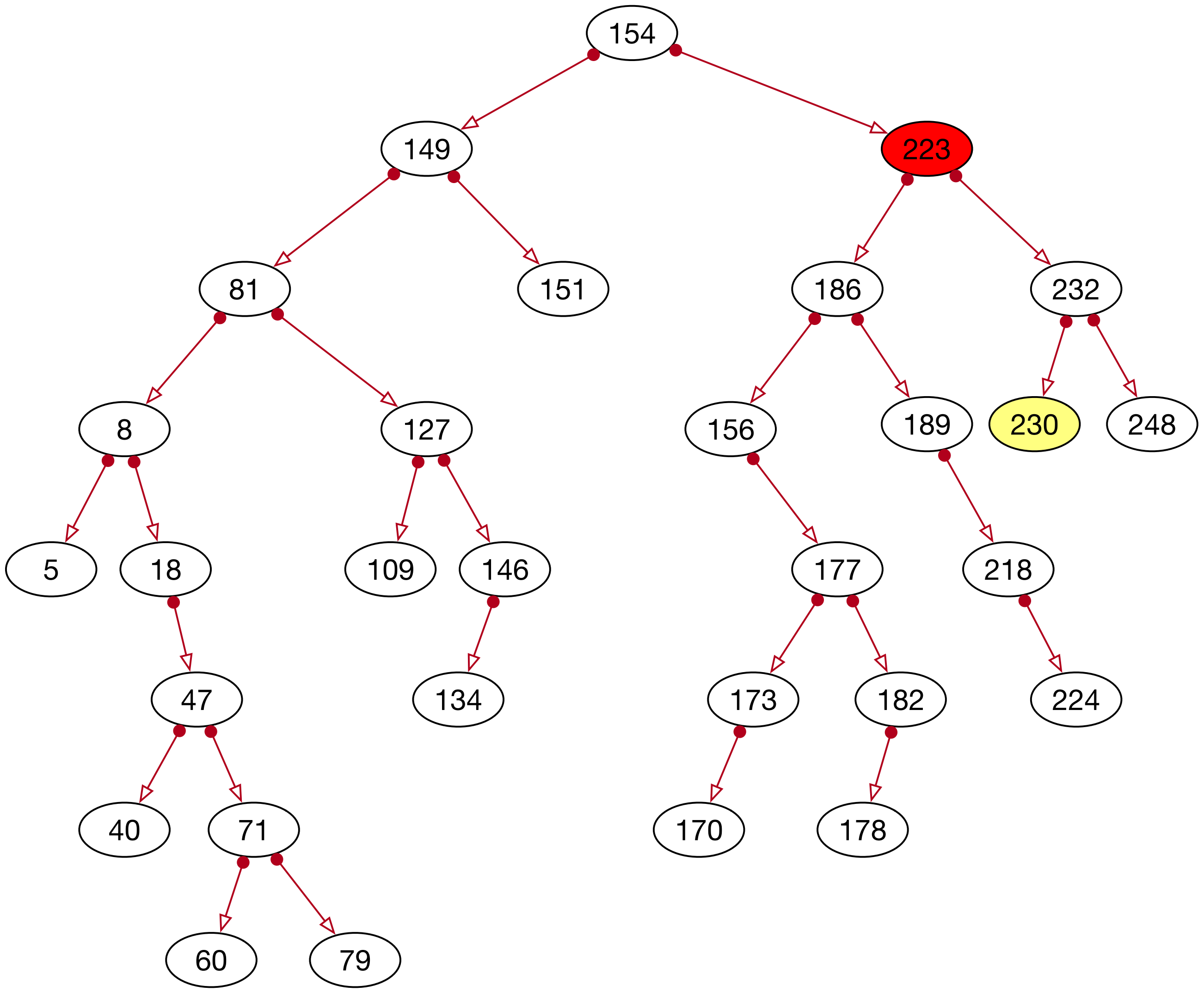
Binary Trees with Parent Link

- 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
```

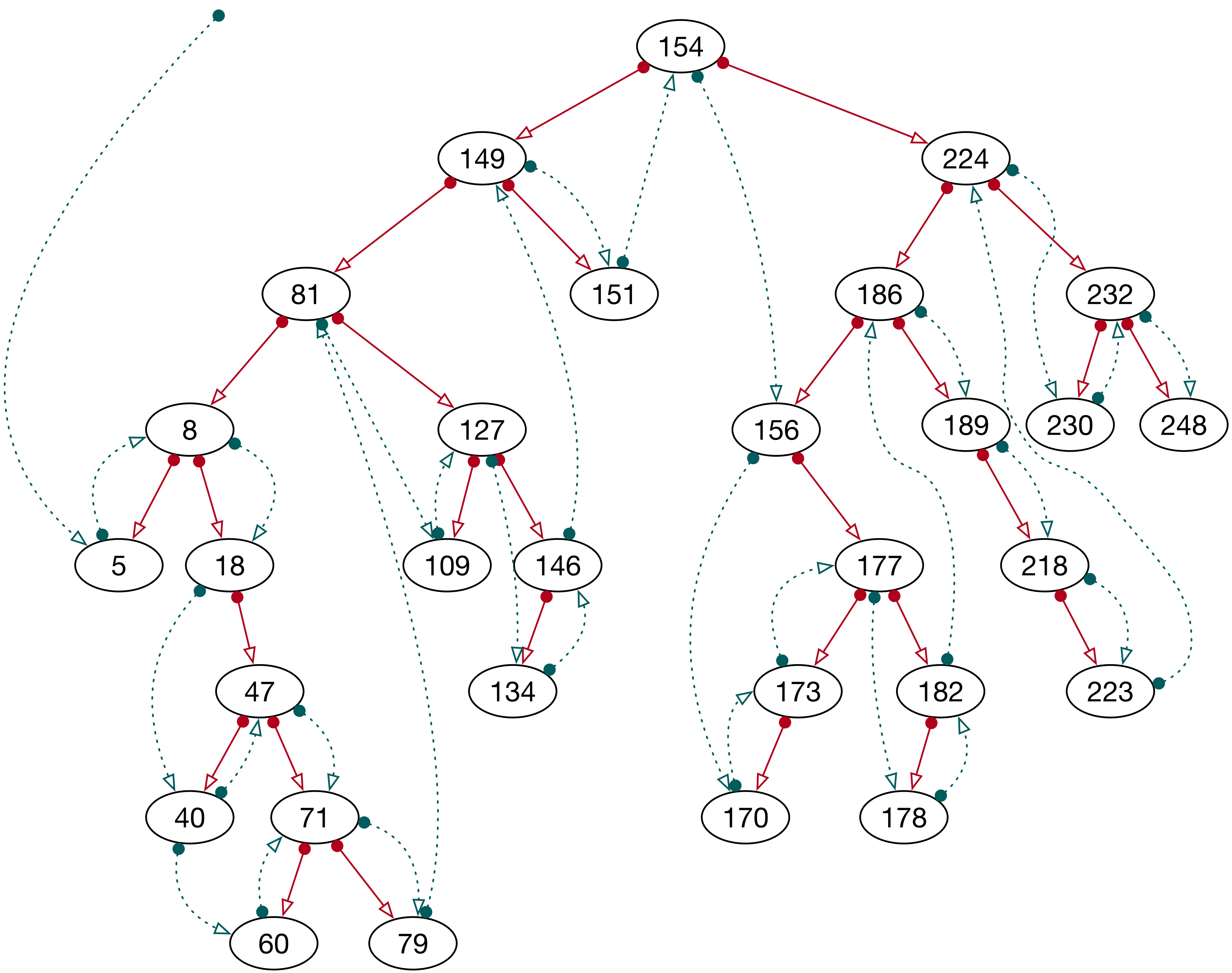
Binary Trees with Parent Link

- 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
```

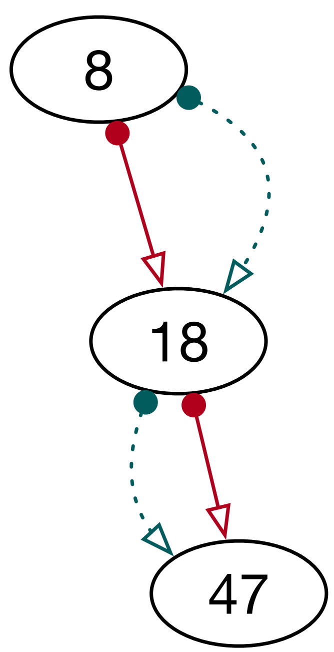
Binary Trees with Parent Link

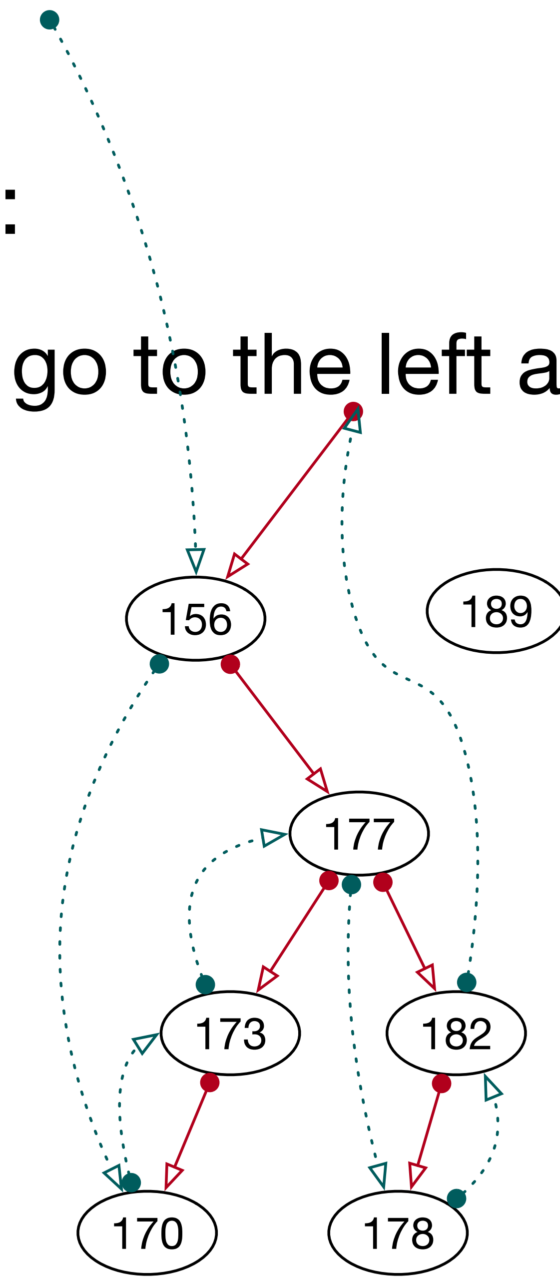
- Non-recursive in-order traversal
  - Here is a tree with an additional set of links for in-order traversal


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

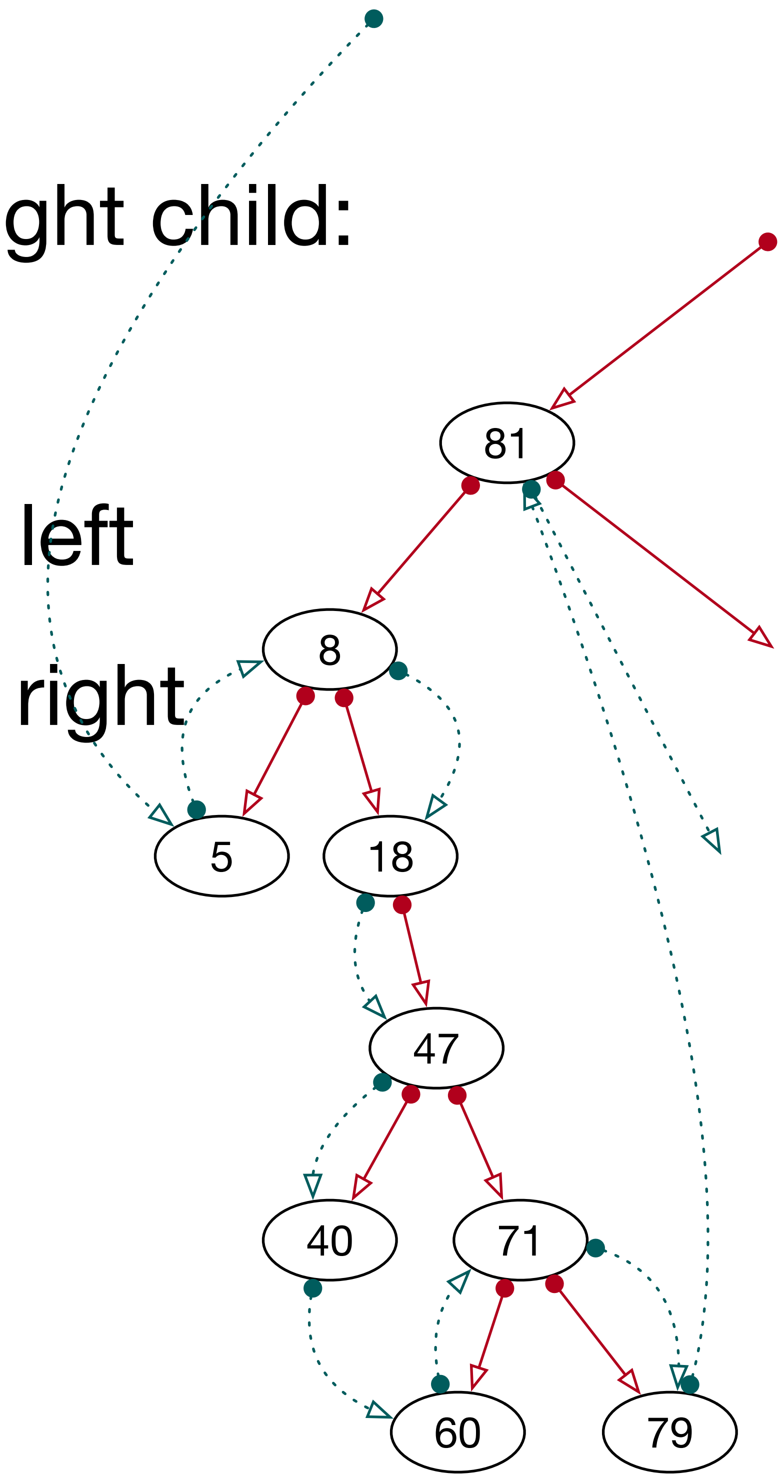
- What is the next node:
  - If the node has a right child:
    - Go one to the right, then go to the left as much as possible





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



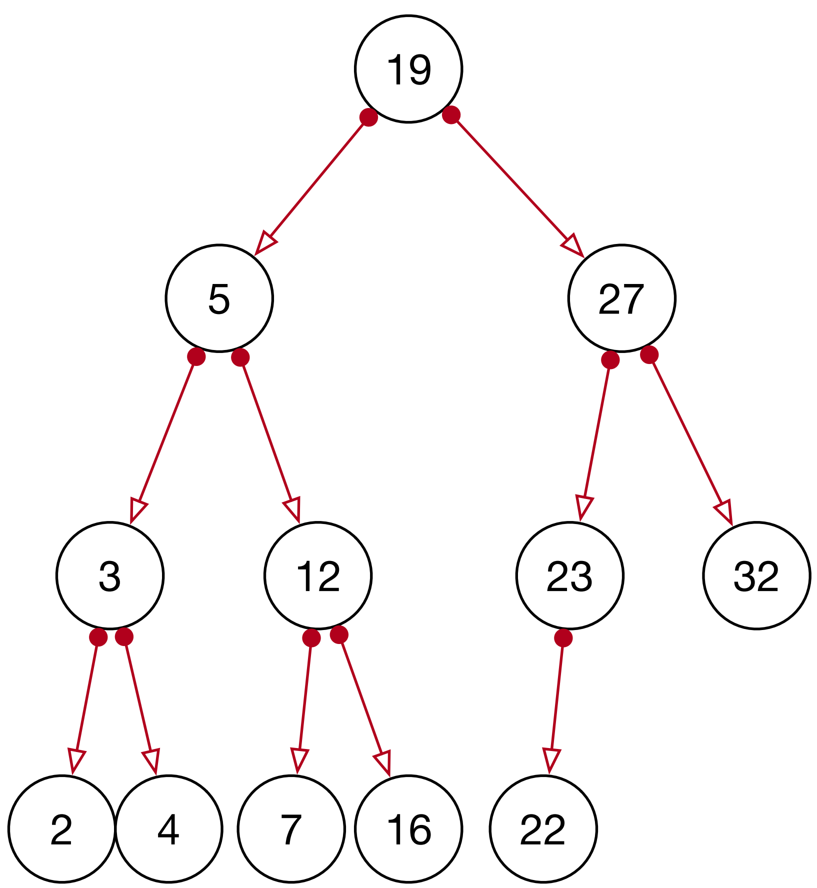
Binary Trees with Parent Link

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

Binary Trees
using
Arrays

Using Arrays

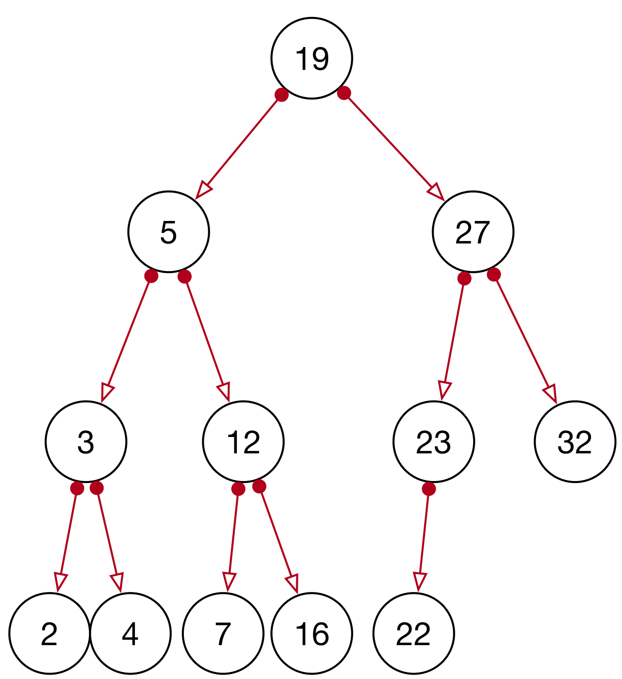
- In a tree, each node has up to two children
  - Can organize nodes in an array
    - Leave first spot open



—	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

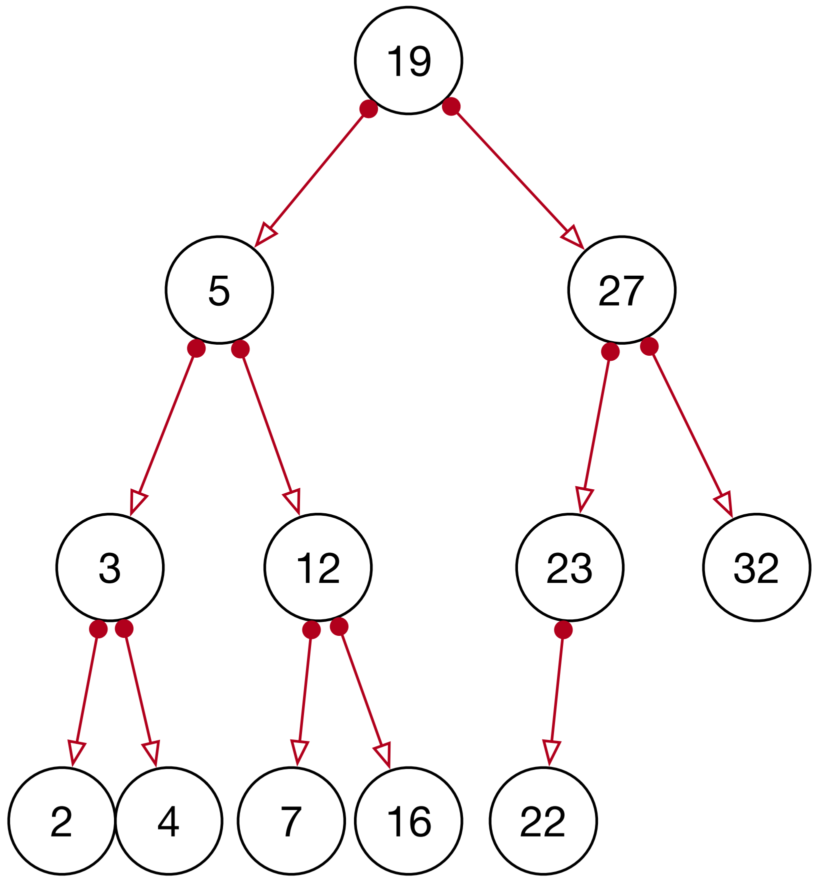
- Left child of node at index i
  - Located at index $2i$
- Right child of node at index i
  - Located at index $2i + 1$



–	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

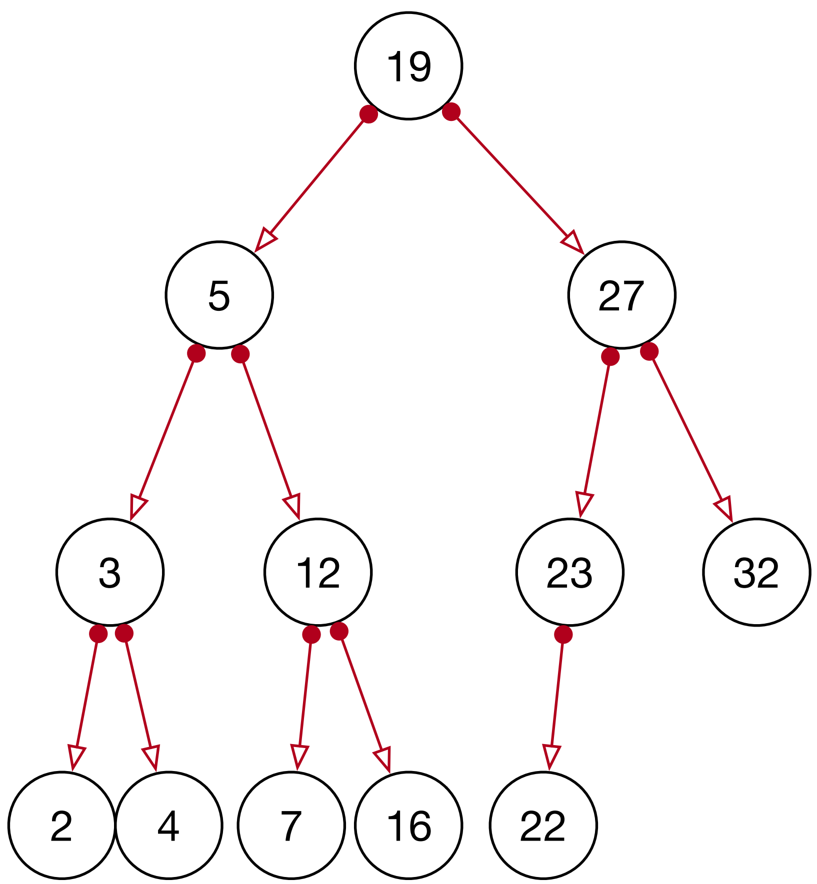
- Parent of node at index i is located at index $i//2$
  - Mathematical notation: $\lfloor \frac{i}{2} \rfloor$



—	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

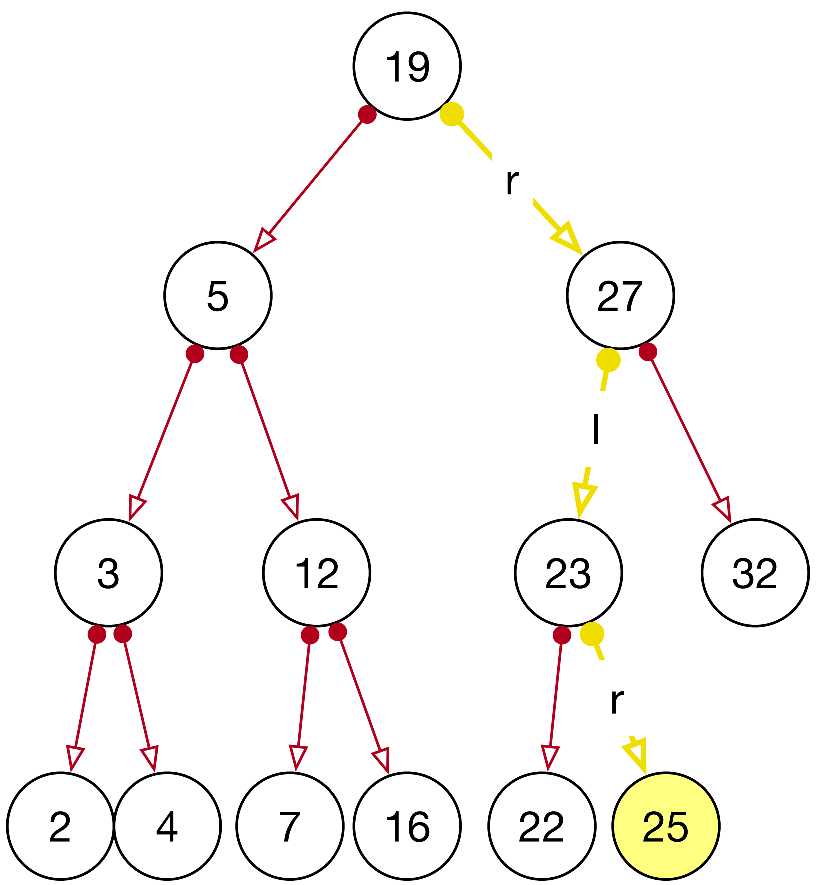
- Right children are at odd indices, left children are even indices



—	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

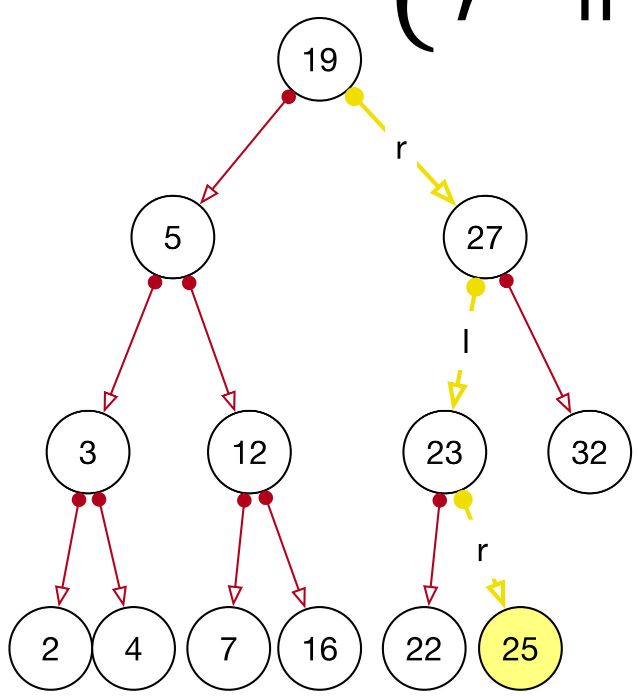


—	19	5	27	3	12	23	32	2	4	7	16	22	25
	1	2	3	4	5	6	7	8	9	10	11	12	13

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

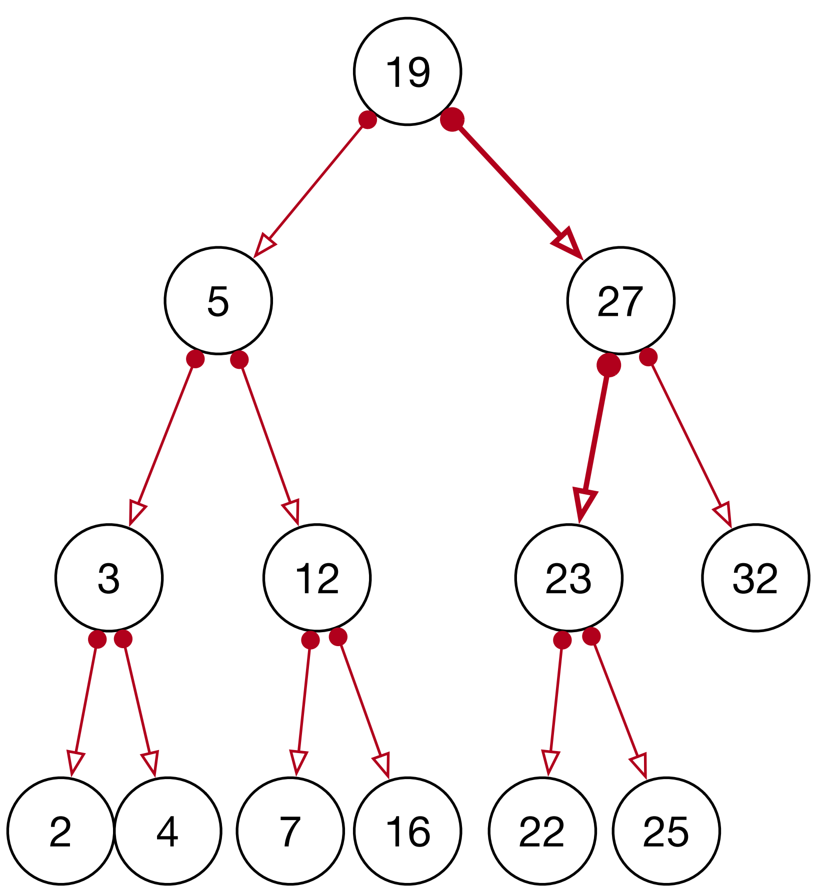
$r \circ l \circ r(1)$

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

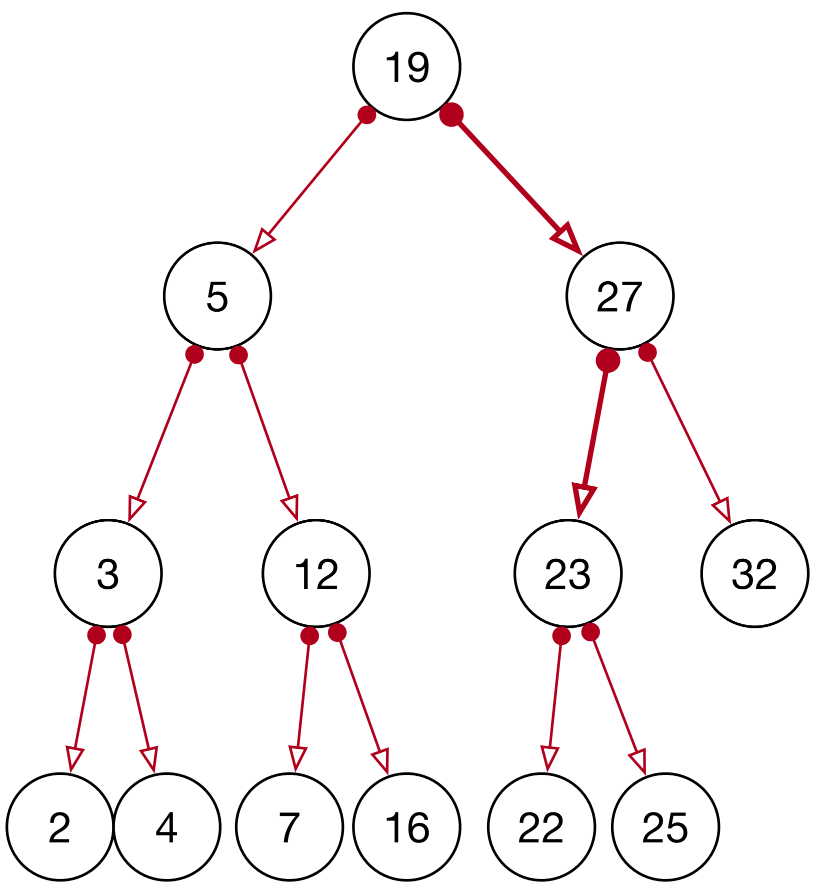
- Children of node i are now $2 \cdot (i+1) - 1 = 2 \cdot i + 1$ and $(2 \cdot (i+1) + 1) - 1 = 2 \cdot i + 2$



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

- ADT 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

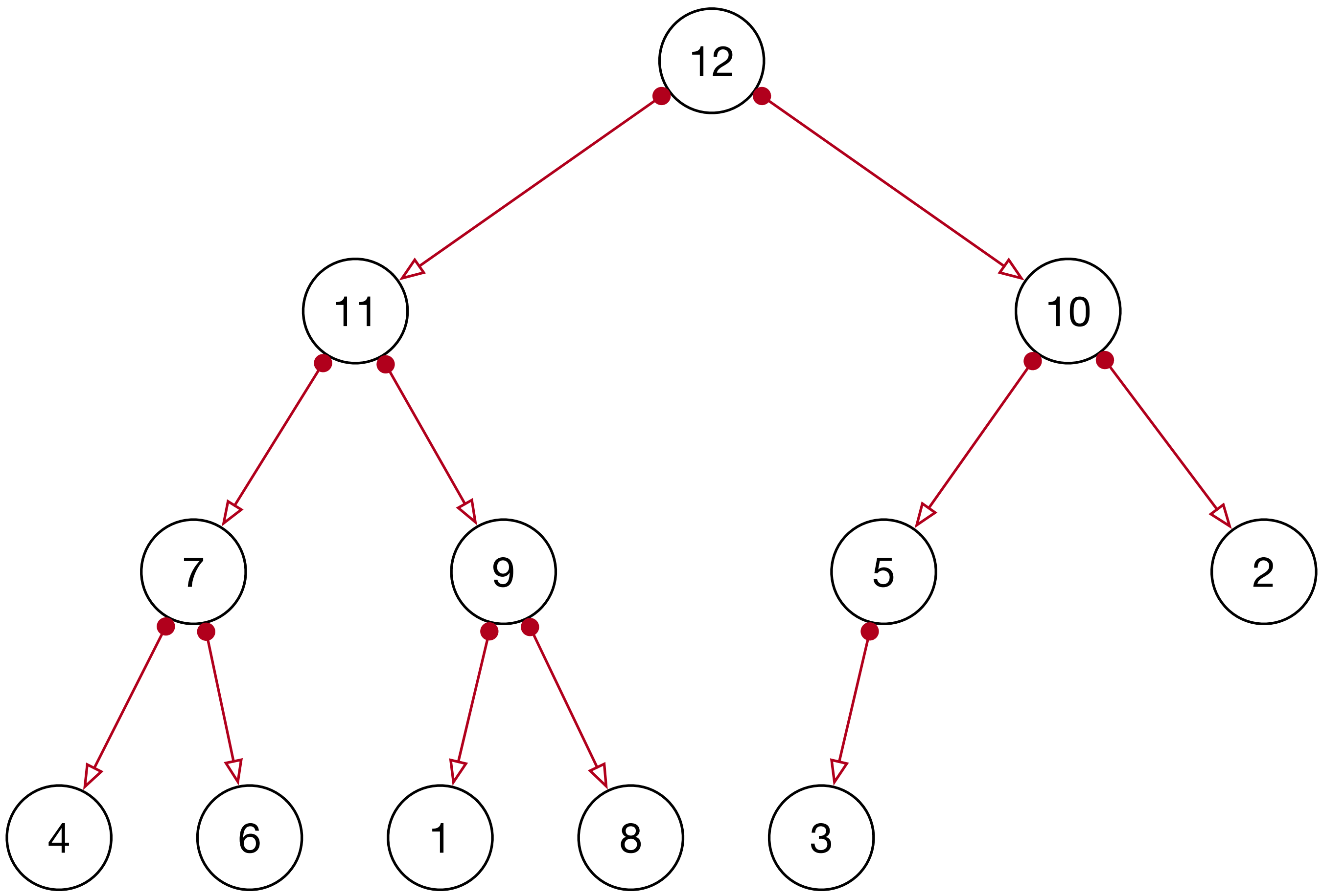
- Simplistic implementation
  - A list
    - Whenever we look for an element, we look for the minimum of the list
    - Run time: Proportional to the length of the list

Priority Queues

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

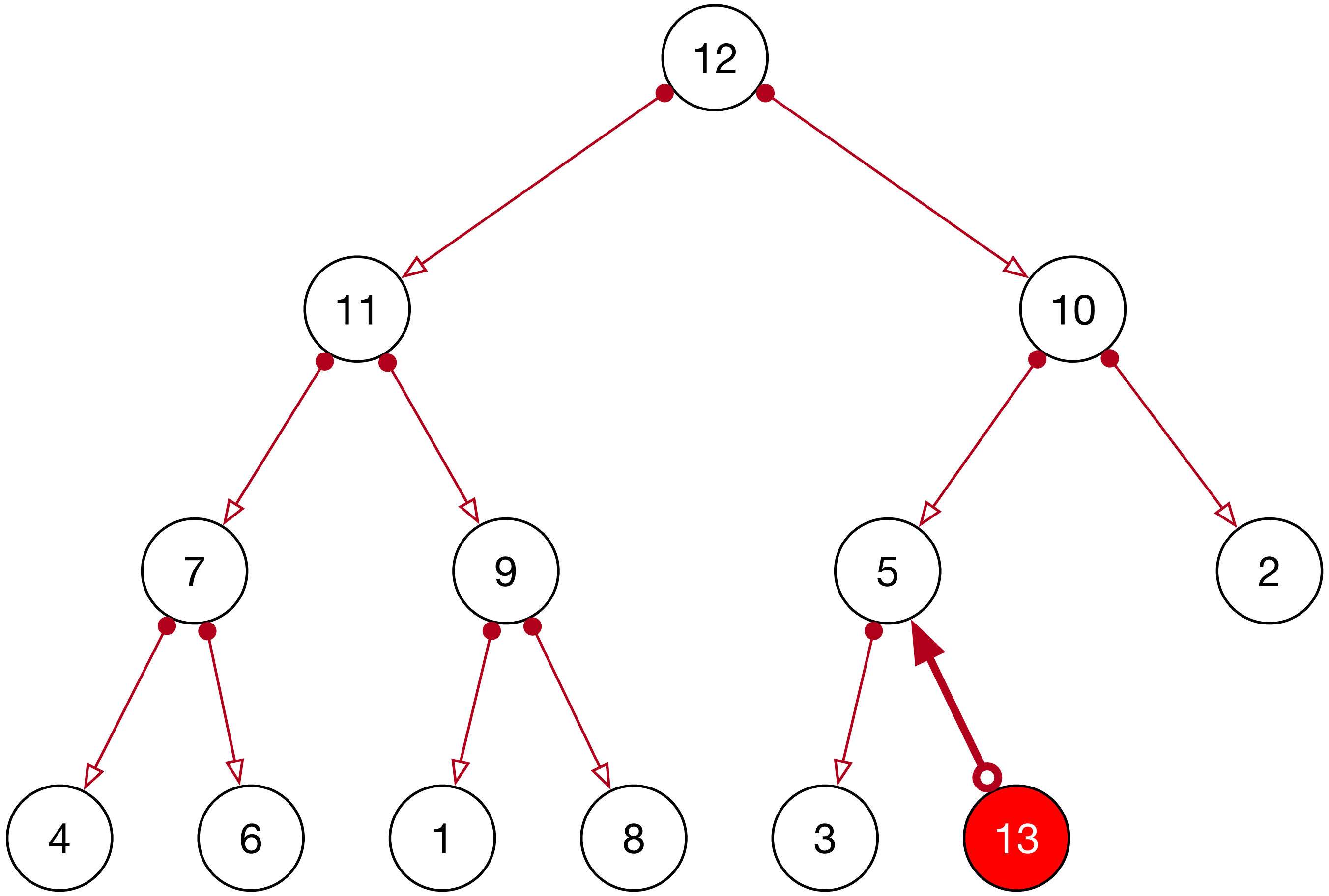
Priority Queues

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



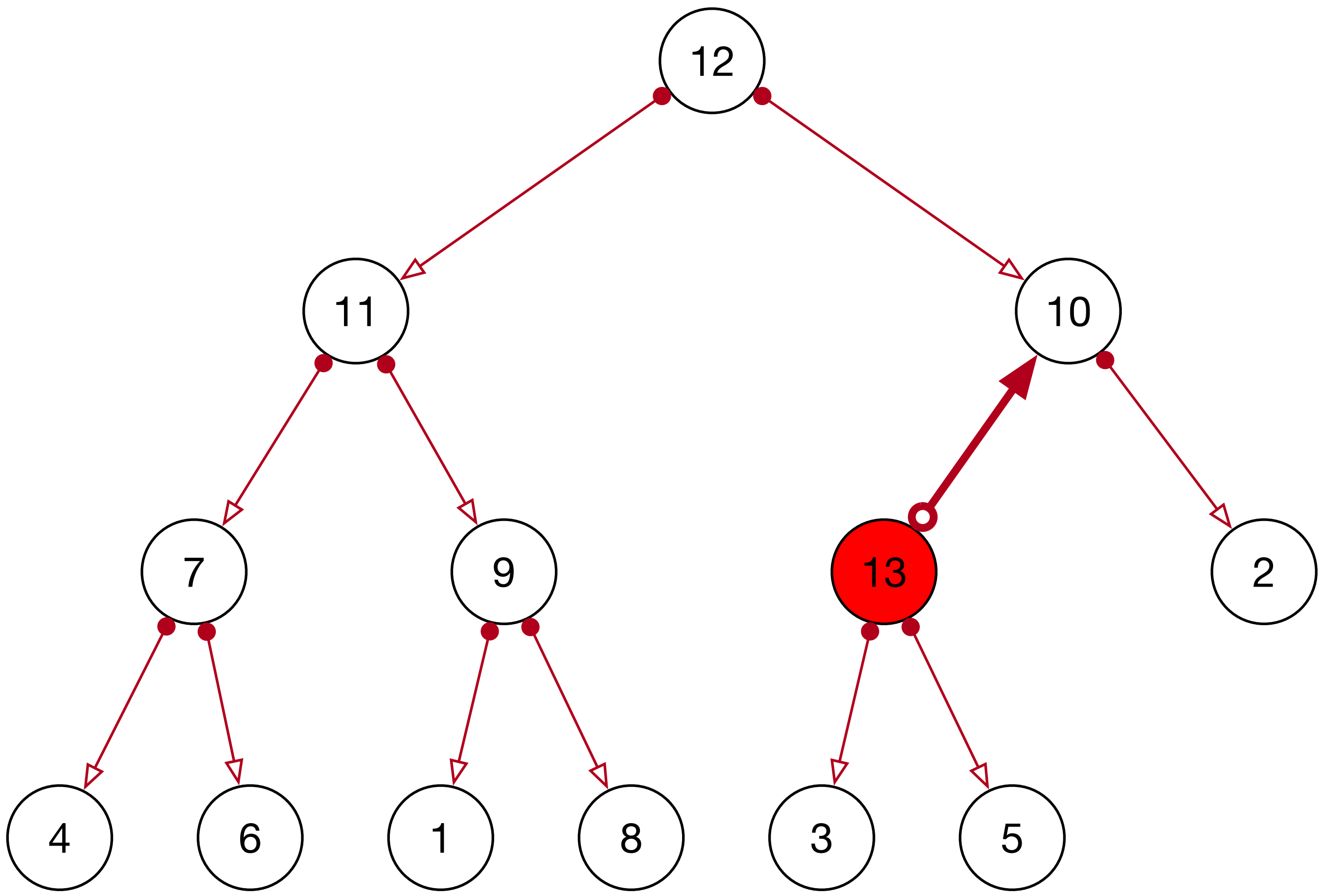
Priority Queues

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



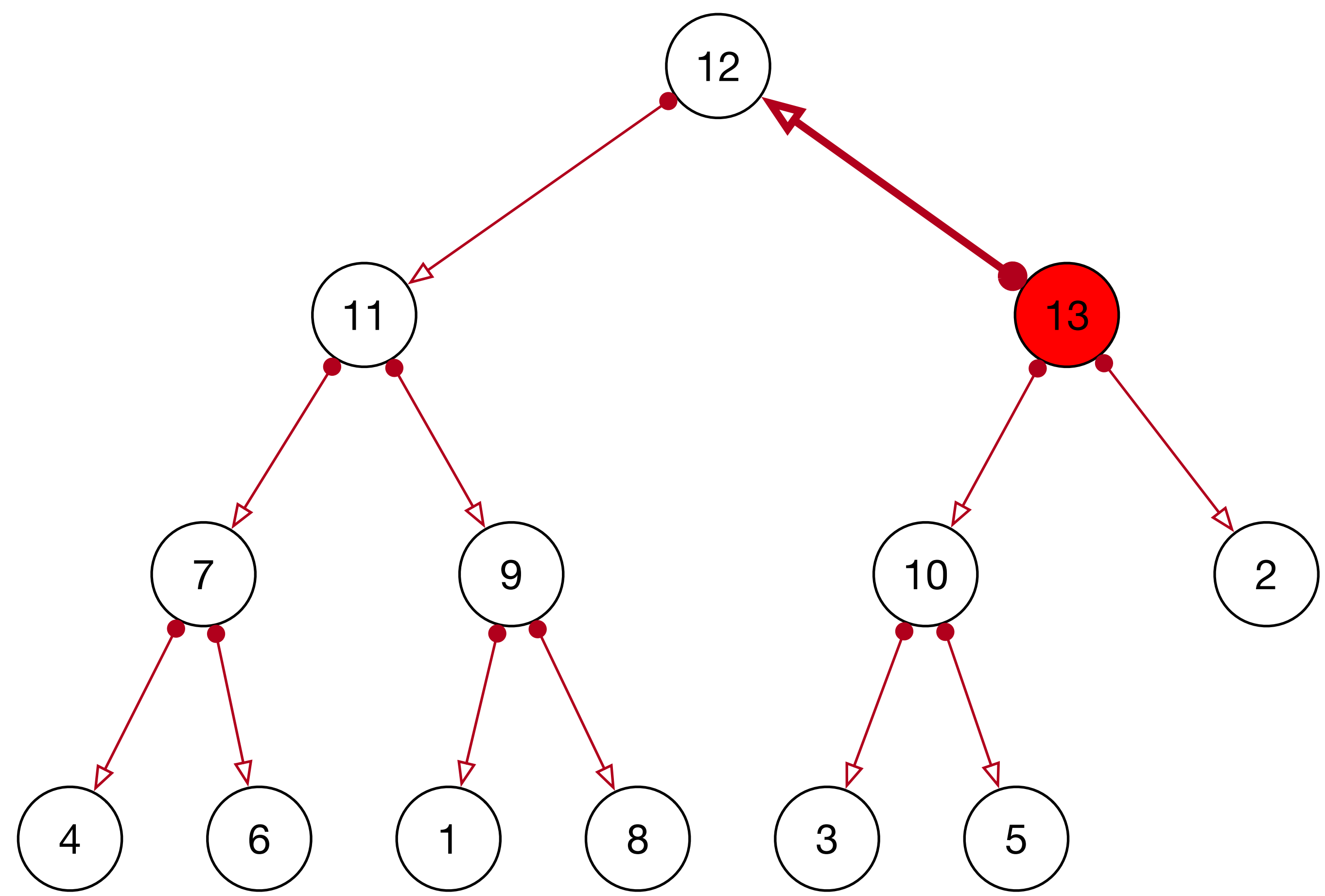
Priority Queues

- This is repeated
  - if necessary



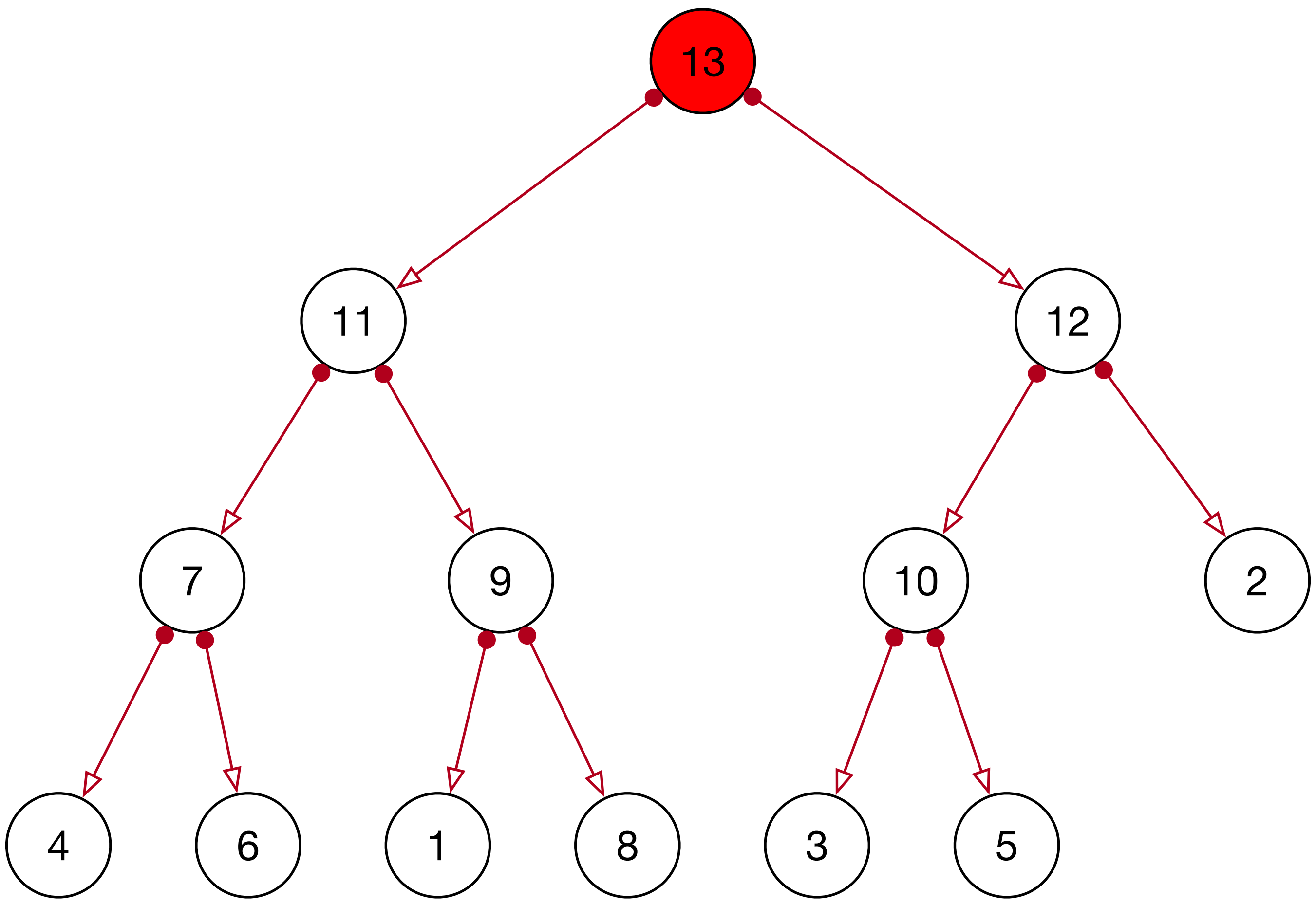
Priority Queues

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



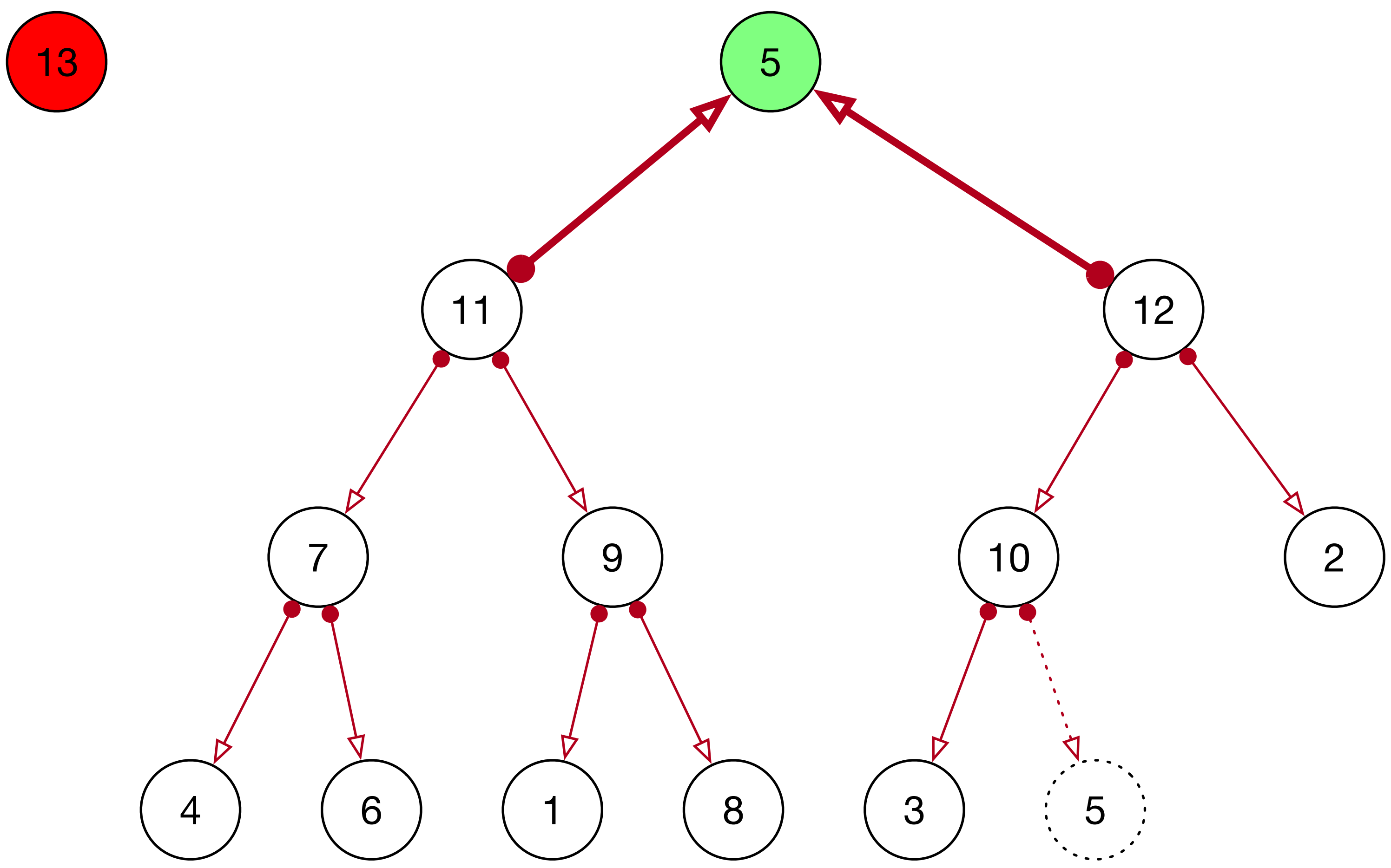
Priority Queues

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



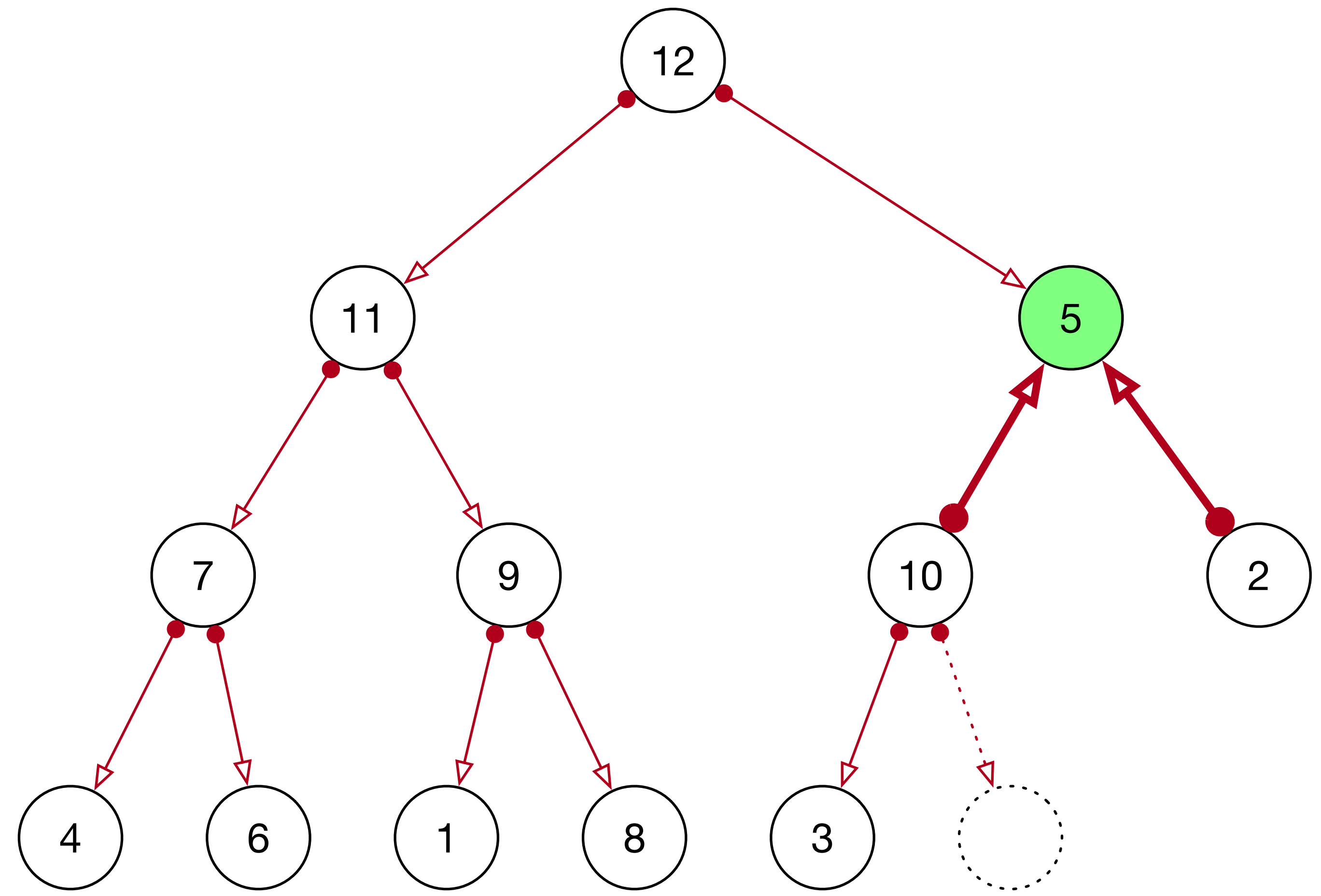
Priority Queues

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



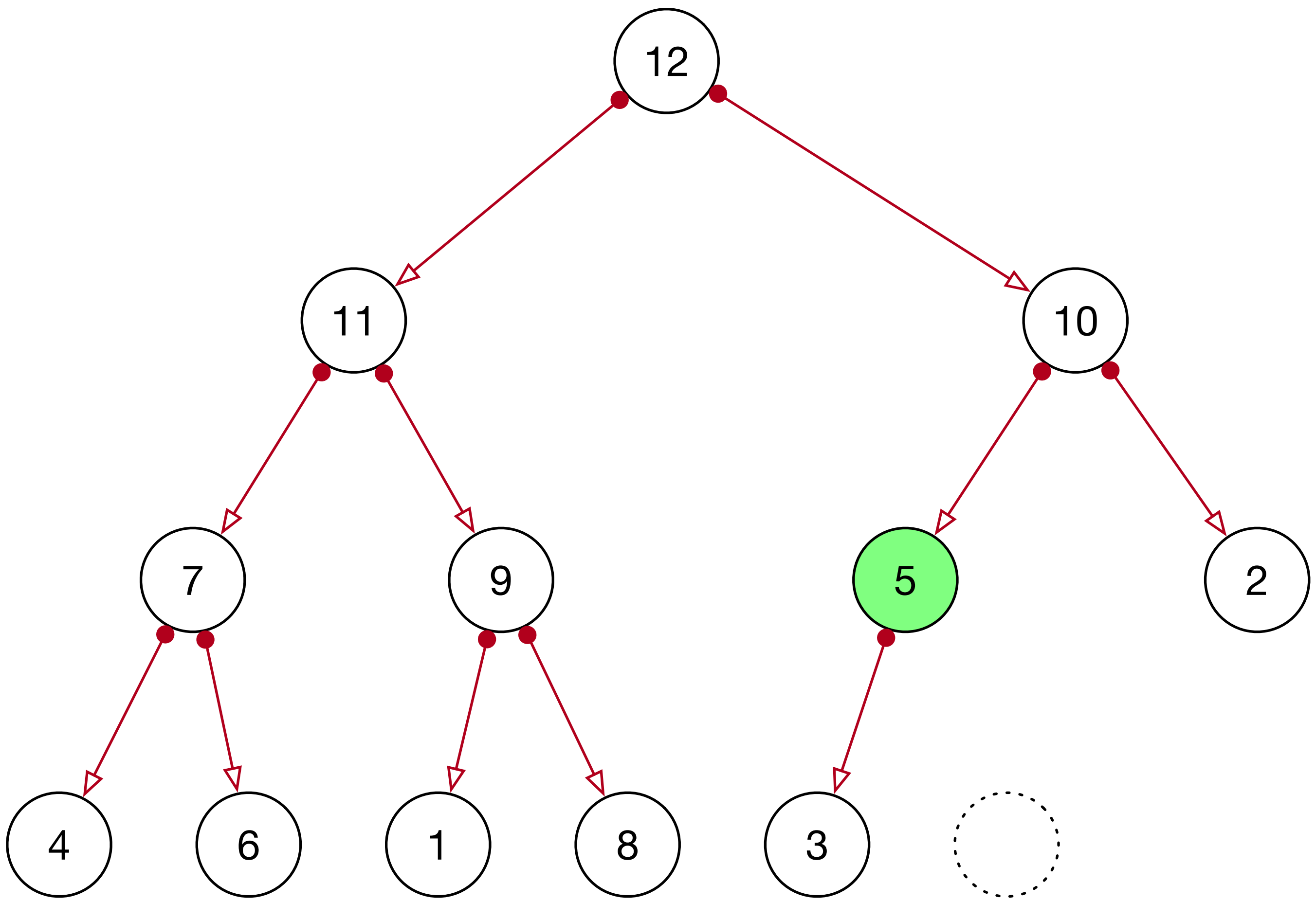
Priority Queues

- 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
```

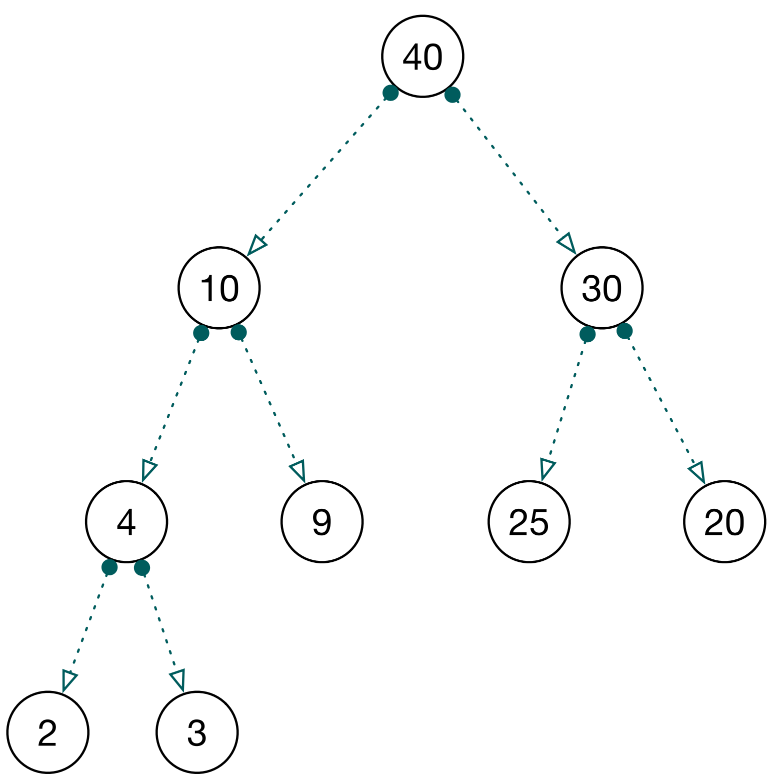
Priority Queues

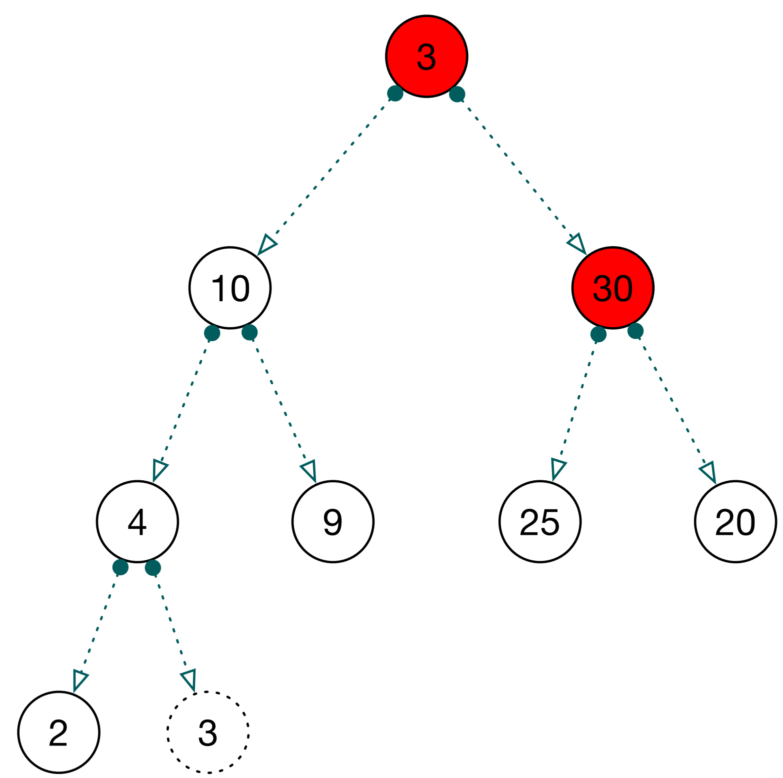
- Then reset the index

```
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

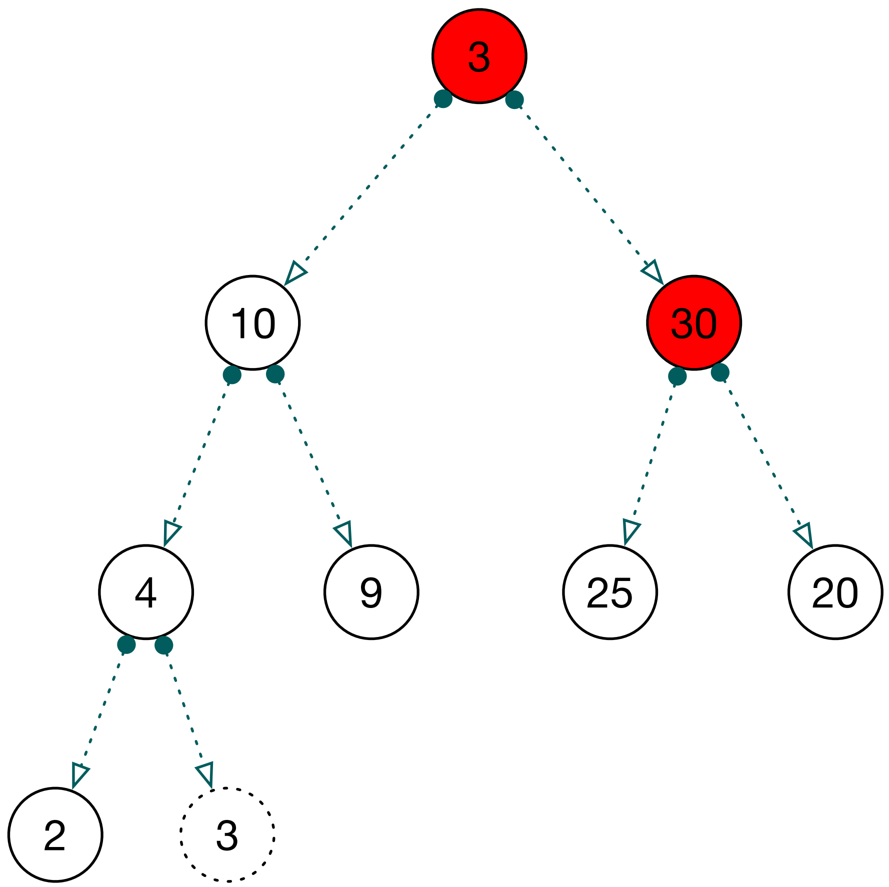
- Extract maximum:
  - Maximum is always at position 0
  - Swap its value with the last element in the array
  - Then heapify:

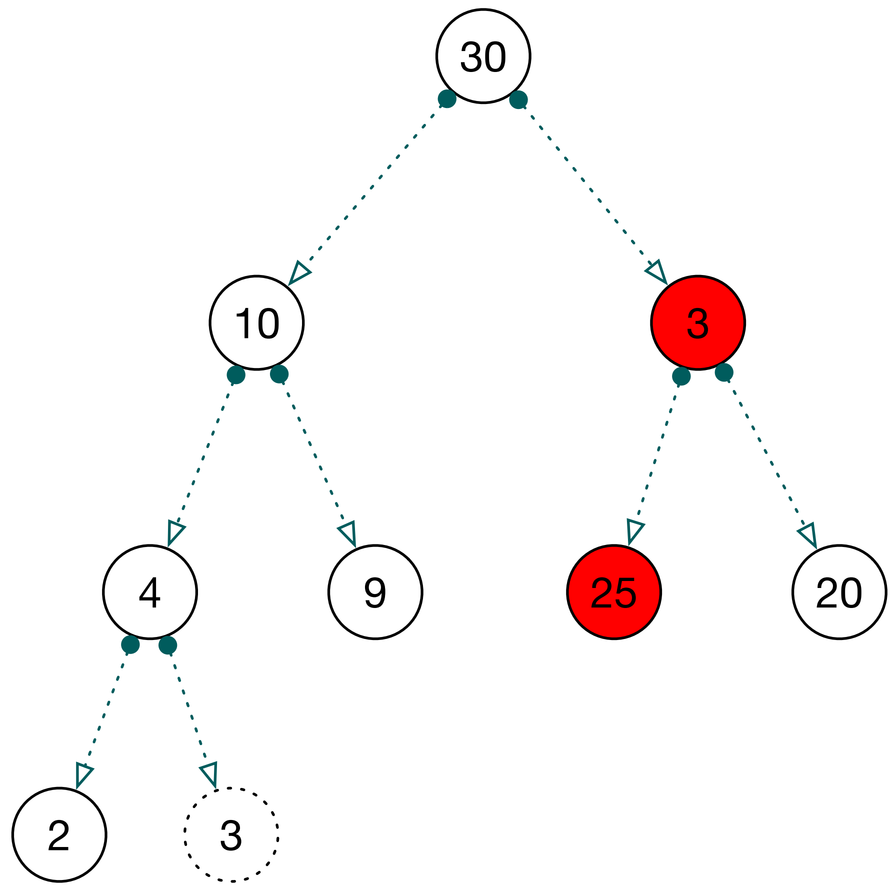




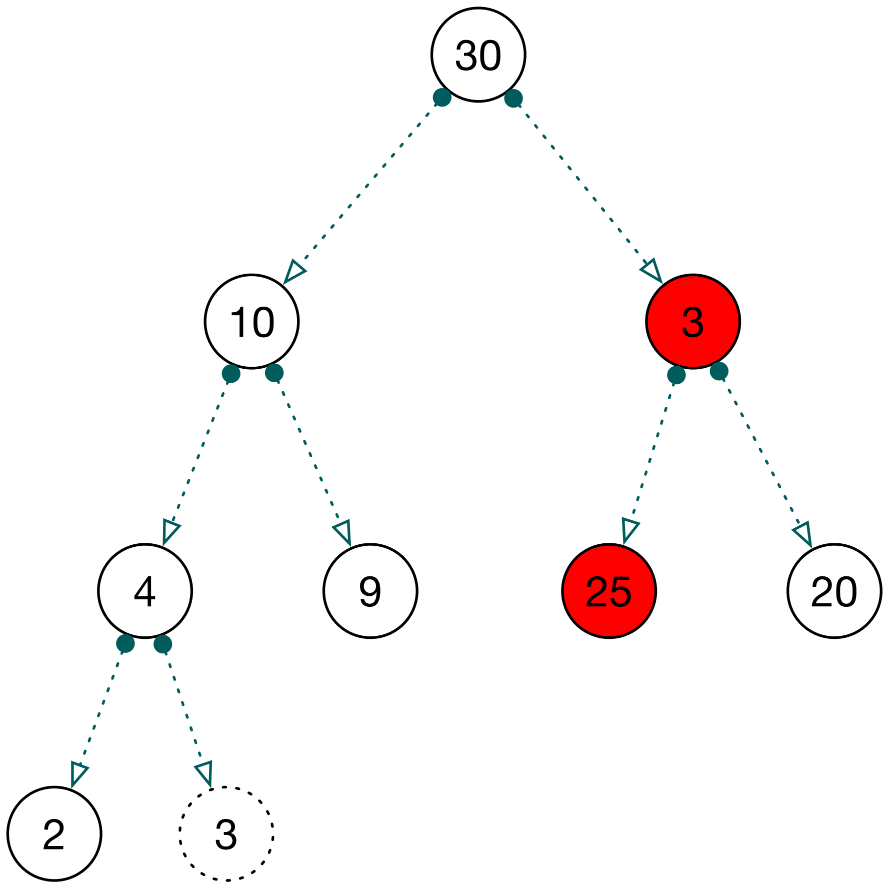
Priority Queues

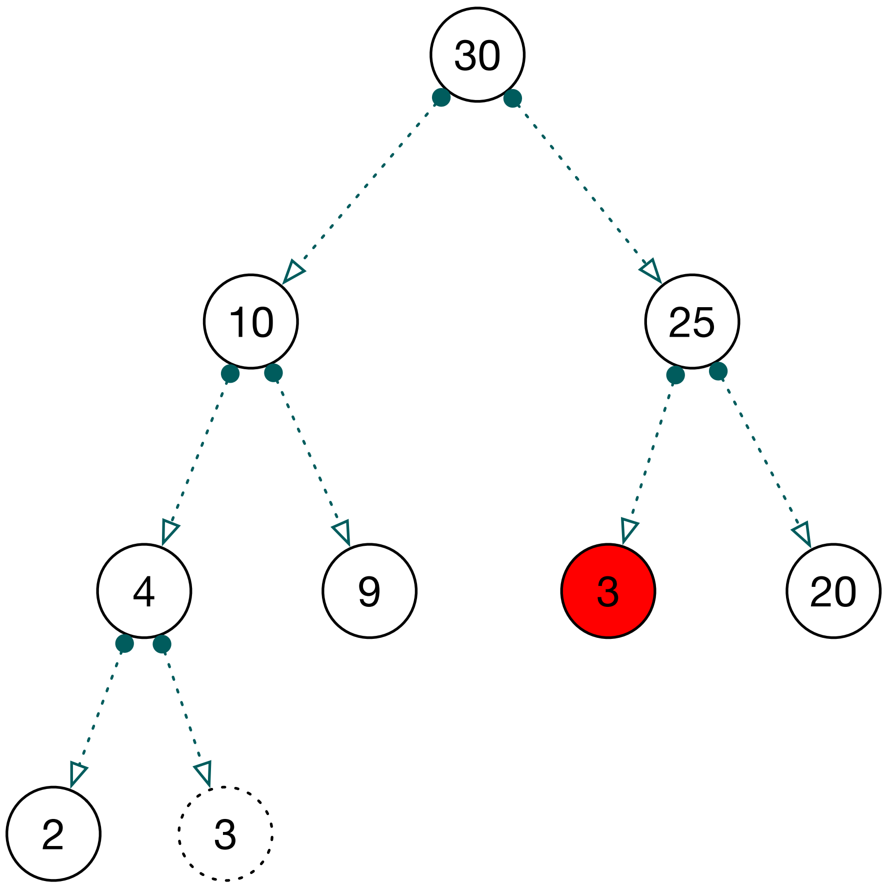
- This is also recursive, but proceeds from top to bottom





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

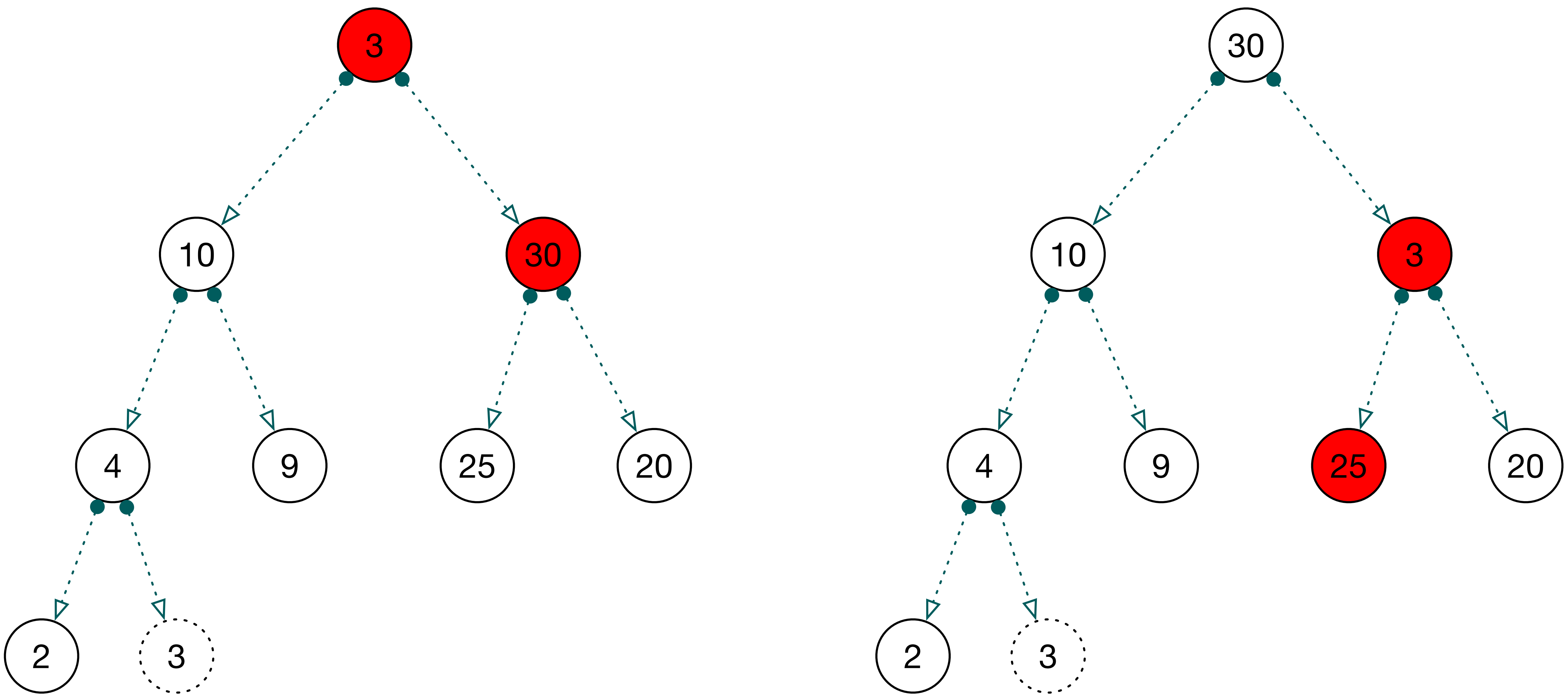
```
 def get_max(self):
     ...
     while n < len(self.array):
          left = PQ.left(n)
          right = PQ.right(n)
          if right < len(self.array):
              if self.array[n] > self.array[left] and
                        self.array[n] > self.array[right]:
                  return ret_val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