Data Structures in Python

- Heap and Priority Queue
- Heap Coding
- **Heap Sort & Min/MaxHeap**

Prof. Youngsup Kim, idebtor@gmail.com, CSEE Dept., Grace School Rm204, Handong Global University

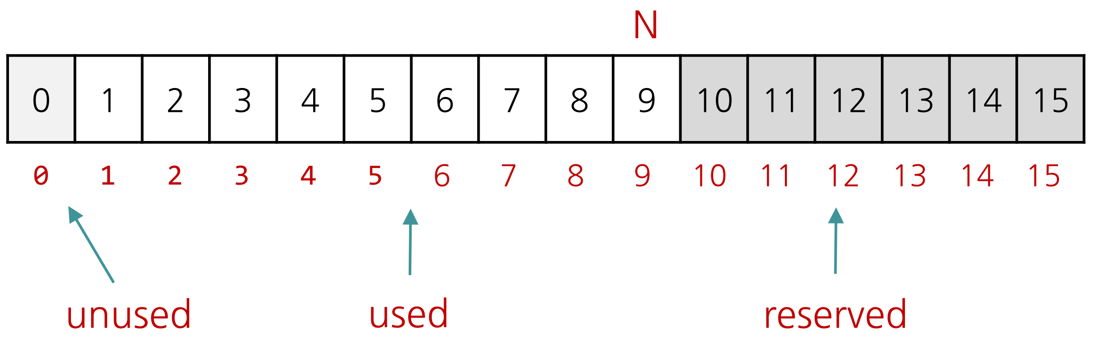
Agenda & Readings

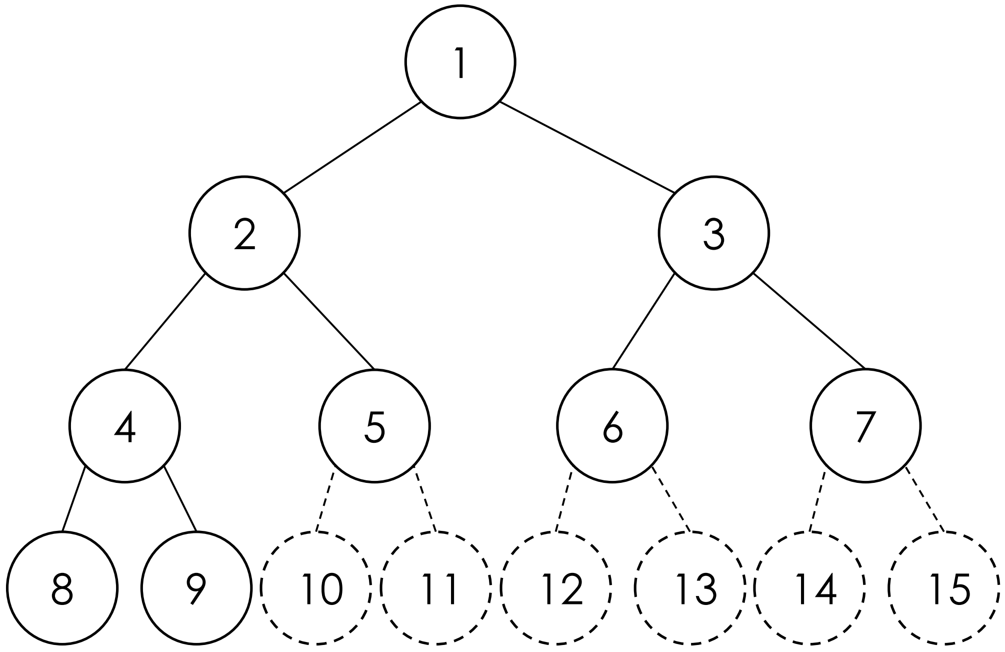
- Min/Max-heap and Heap sort
  - Min-heap and max-heap
  - Min-heap and max-heap conversion
  - Heap sort
  - Time Complexity

- Reference:
  - Problem Solving with Algorithms and Data Structures

Binary trees - Array representation

- A **complete** binary tree with n nodes, any node index i, $1 \le i \le n$, we have
  - `parent(i) is at` $\lfloor i/2 \rfloor$ `If i = 1, i is at the root and has no parent`
  - `leftChild(i) is at 2i  if 2i <= n. If 2i > n, then i has no left child.`
  - `rightChild(i) is at 2i+1 if 2i+1 <= n. If 2i+1 > n, then i has no right child.`



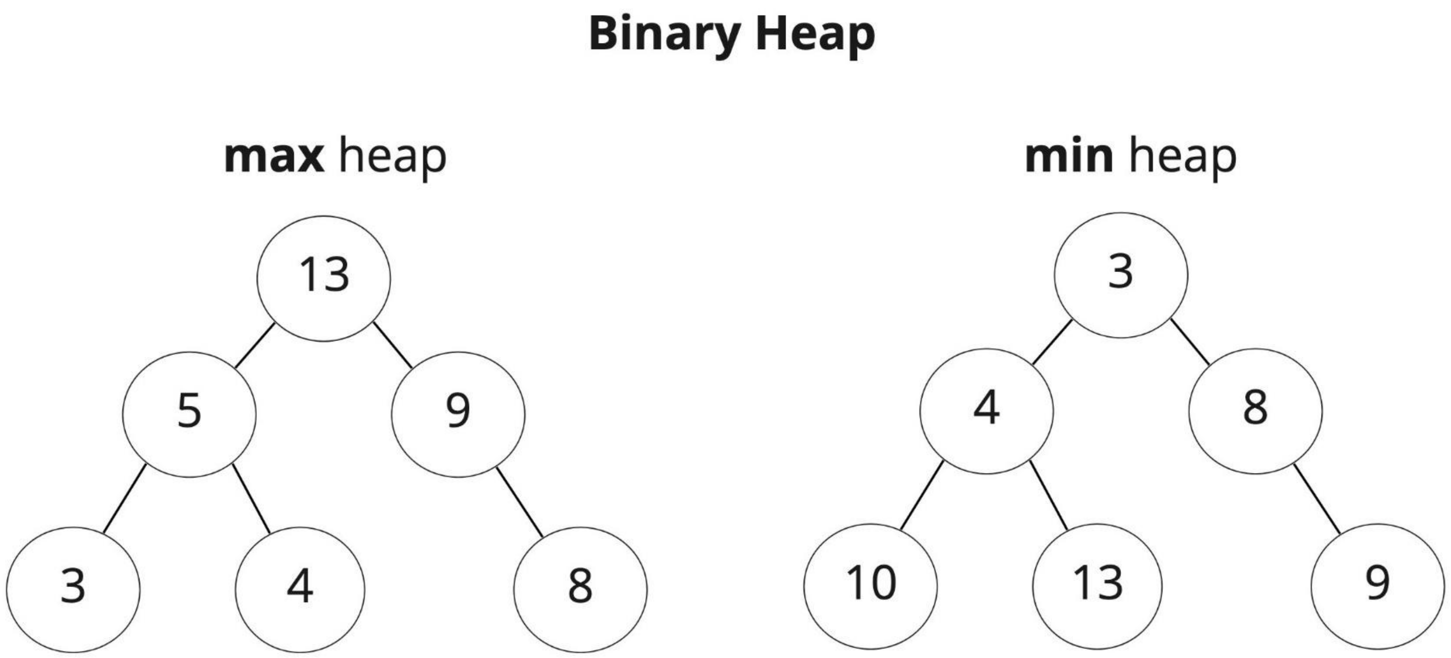


*A **complete** binary tree*

Min/Max-Heap Conversion

Algorithm

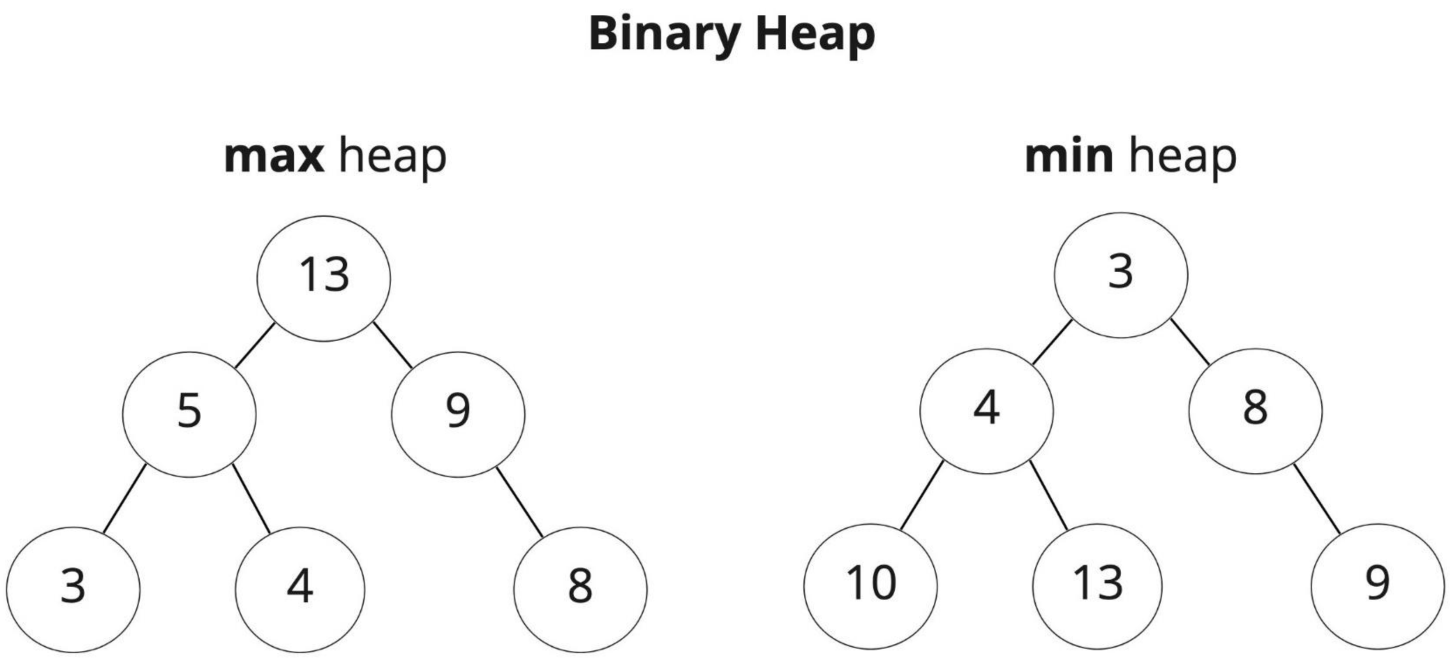
- Change the comparator
- heapify()



Min/Max-Heap Conversion

Algorithm

- Change the comparator: **Where and How?**
- heapify()



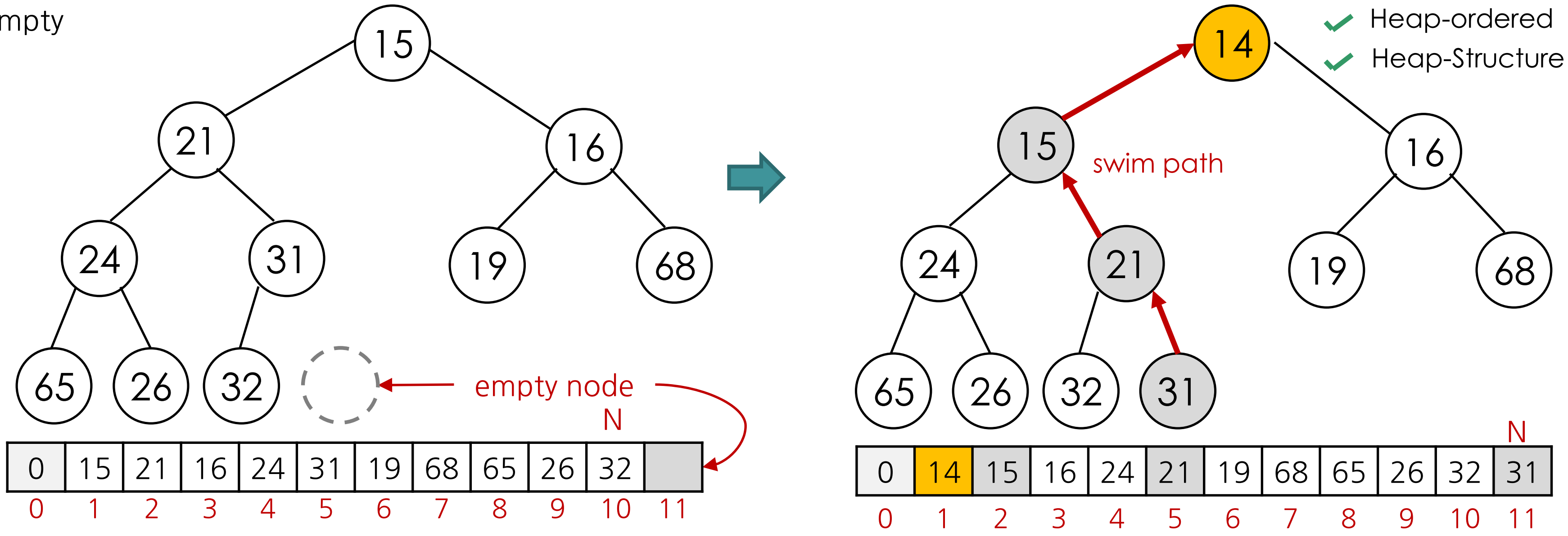
min-heap: insert(heap, 14)

Algorithm:

- Insert a new element **while maintaining a heap-structure**
- **swim():** Move the element up the heap **while not satisfying heap-ordered**

```
class BinHeap:
    ...
    def insert(self, key):                   # check N and len(heap) before using
        self.heap.append(key)                # append() if necessary, otherwise use list index
        self.N += 1
        self.swim(self.N)
```

Where is an empty node to start?

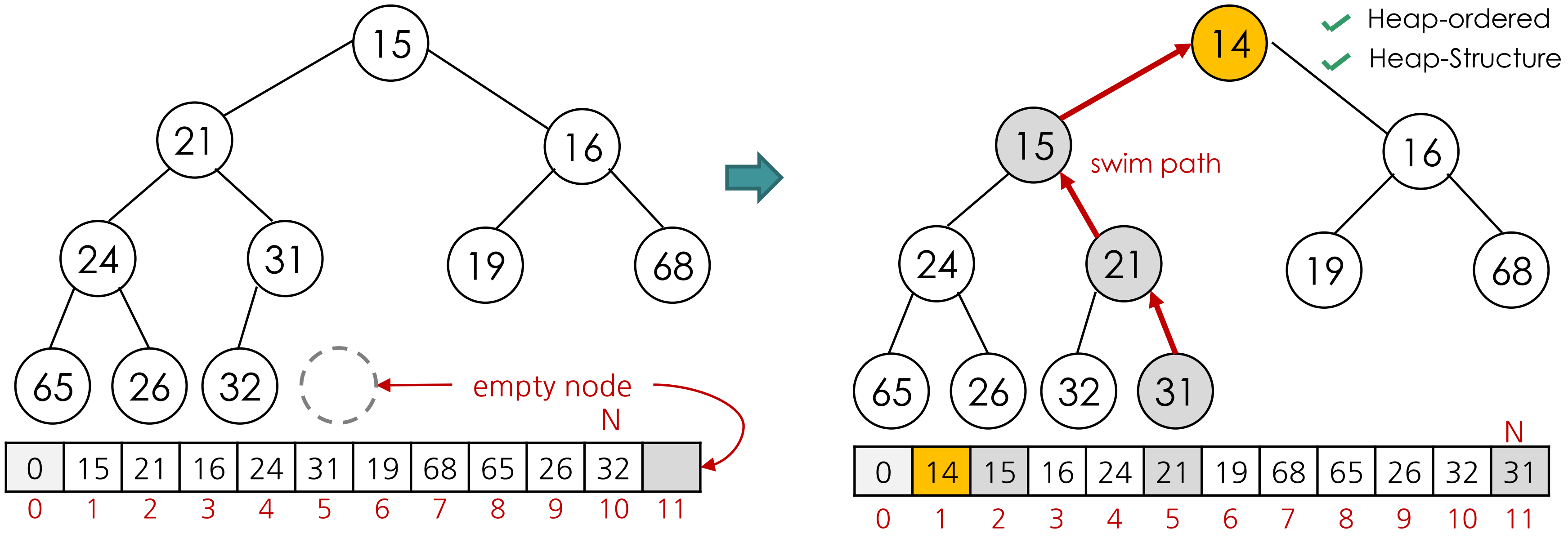


min-heap: insert(heap, 14)

Algorithm:

- Insert a new element **while maintaining a heap-structure**
- swim(): Move the element up the heap **while not satisfying heap-ordered**

```
class BinHeap:
    ...
    def swim(self, k):                           # append key and swim up
        while k // 2 > 0:                        # if not reached root
            if self.heap[k//2] > self.heap[k]:   # if parent is more than kid (minheap)
                self.swap(k//2, k)               # swap(parent, kid)
            k = k // 2                           # swim up - move to the parent node
```



Min/Max-Heap Conversion

Algorithm

- Change the comparator: **Where and How?**
- heapify()

```
class BinHeap:
    ...
    def swim(self, k):                              # append key and swim up
        while k // 2 > 0:                           # if not reached root
            if self.heap[k//2] > self.heap[k]:      # if parent is more than kid (minheap)
                self.swap(k//2, k)                  # swap(parent, kid)
            k = k // 2                              # swim up - move to the parent node
```

```
class BinHeap:
    ...
    def sink(self, i):                                 # start sink at node i
        while (i * 2) <= self.N:                       # not bottom of tree yet?
            k = 2 * i                                  # left child
            if k < self.N and self.heap[k] > self.heap[k+1]:  # select one of two kids to compare
                k += 1                                 # right child is selected
            if not self.heap[i] > self.heap[k]: break  # break if node i and kid are heap-ordered
            self.swap(i, k)                            # if not heap-ordered, swap i and k
            i = k                                      # i becomes k & continue sink process
```

Min/Max-Heap Conversion

Algorithm

- Change the comparator: **Where and How?**
- heapify()

```
class BinHeap:
    ...
    def swim(self, k):                          # append key and swim up
        while k // 2 > 0:                       # if not reached root
            if self.heap[k//2] > self.heap[k]:  # if parent is more than kid (minheap)
                self.swap(k//2, k)              # swap(parent, kid)
            k = k // 2                          # swim up - move to the parent node
```

```
class BinHeap:
    ...
    def sink(self, i):                                   # start sink at node i
        while (i * 2) <= self.N:                         # not bottom of tree yet?
            k = 2 * i                                    # left child
            if k < self.N and self.heap[k] > self.heap[k+1]:  # select one of two kids to compare
                k += 1                                   # right child is selected
            if not self.heap[i] > self.heap[k]: break    # break if node i and kid are heap-ordered
            self.swap(i, k)                              # if not heap-ordered, swap i and k
            i = k                                        # i becomes k & continue sink process
```

Min/Max-Heap Conversion

Algorithm

- Change the comparator: **Where and How?**
- heapify()

```
def less(self, p, k):                    # p: parent, k: kid
    return self.heap[p] < self.heap[k]   # comparator

def more(self, p, k):                    # p: parent, k: kid
    return self.heap[p] > self.heap[k]   # comparator
```

```
class BinHeap:
    ...
    def swim(self, k):                         # append key and swim up
        while k // 2 > 0:                     # if not reached root
            if self.heap[k//2] > self.heap[k]: # if parent is more than kid (minheap)
                self.swap(k//2, k)            # swap(parent, kid)
            k = k // 2                        # swim up - move to the parent node
```

```
class BinHeap:
    ...
    def sink(self, i):                                   # start sink at node i
        while (i * 2) <= self.N:                          # not bottom of tree yet?
            k = 2 * i                                     # left child
            if k < self.N and self.heap[k] > self.heap[k+1]:  # select one of two kids to compare
                k += 1                                    # right child is selected
            if not self.heap[i] > self.heap[k]: break     # break if node i and kid are heap-ordered
            self.swap(i, k)                               # if not heap-ordered, swap i and k
            i = k                                         # i becomes k & continue sink process
```

Min/Max-Heap Conversion

Algorithm

- Change the comparator: **Where and How?**
- heapify()

```
def less(self, p, k):                  # p: parent, k: kid
    return self.heap[p] < self.heap[k]  # comparator

def more(self, p, k):                  # p: parent, k: kid
    return self.heap[p] > self.heap[k]  # comparator
```

```
class BinHeap:
    ...
    def swim(self, k):                        # append key and swim up
        while k // 2 > 0:                     # if not reached root
            if self.heap[k//2] > self.heap[k]: # if parent is more than kid (minheap)
                self.swap(k//2, k)            # swap(parent, kid)
            k = k // 2                        # swim up - move to the parent node
```

```
self.more(k//2, k)
```

```
class BinHeap:
    ...
    def sink(self, i):                        # start sink at node i
        while (i * 2) <= self.N:              # not bottom of tree yet?
            k = 2 * i                         # left child
            if k < self.N and self.heap[k] > self.heap[k+1]:  # select one of two kids to compare
                k += 1                        # right child is selected
            if not self.heap[i] > self.heap[k]: break # break if node i and kid are heap-orde
            self.swap(i, k)                   # if not heap-ordered, swap i and k
            i = k                             # i becomes k & continue sink process
```

```
self.more(k, k + 1)
```

```
self.more(i, k)
```

Min/Max-Heap Conversion

Algorithm

- Change the comparator: **Where and How?**
- heapify()

```
class BinHeap:
    def __init__(self):                    # min-heap by default
        self.heap = [0]                    # list, index 0 is not used
        self.N = 0                         # points the last valid heap element index
        self.comp = None                   # used later, comparator to switch between min-heap & max-heap
```

```
class BinHeap:
    def __init__(self, min = True):        # min-heap by default
        self.heap = [0]                    # list, index 0 is not used
        self.N = 0                         # points the last valid heap element index
        self.comp = self.more if min else self.less    # set comparator, more for minheap, less for max
```

Min/Max-Heap Conversion

Algorithm

- Change the comparator: **Where and How?**
- heapify()

```
def less(self, p, k):                        # p: parent, k: kid
    return self.heap[p] < self.heap[k]       # comparator

def more(self, p, k):                        # p: parent, k: kid
    return self.heap[p] > self.heap[k]       # comparator
```

```
class BinHeap:
    ...
    def swim(self, k):                       # append key and swim up
        while k // 2 > 0:                    # if not reached root
            if self.heap[k//2] > self.heap[k]:   # if parent is more than kid (minheap)
                self.swap(k//2, k)           # swap(parent, kid)
            k = k // 2                       # swim up - move to the parent node
```

`self.comp(k//2, k)`

```
class BinHeap:
    ...
    def sink(self, i):                                 # start sink at node i
        while (i * 2) <= self.N:                       # not bottom of tree yet?
            k = 2 * i                                  # left child
            if k < self.N and self.heap[k] > self.heap[k+1]:  # select one of two kids to compare
                k += 1                                 # right child is selected
            if not self.heap[i] > self.heap[k]: break  # break if node i and kid are heap-orde
            self.swap(i, k)                            # if not heap-ordered, swap i and k
            i = k                                      # i becomes k & continue sink process
```

`self.comp(k, k + 1)`

`self.comp(i, k)`

Min/Max-Heap Conversion

Algorithm

- Change the comparator: **Where and How?**
- heapify()

```
def buildHeap(self, arr, min = None):
    if min == True:                     # set min-heap comparator
        self.comp = self.more
    elif min == False:                  # set max-heap comparator
        self.comp = self.less           # None: no change

    self.heap = [0] + arr[:]
    self.N = len(arr)
    i = len(arr) // 2                   # begin with the last internal node
    while i > 0:
        self.sink(i)
        i -= 1

def heapify(self, min = None):
    self.buildHeap(self.heap[1:], min)
```

Min/Max-Heap Conversion

Algorithm

- Change the comparator: **Where and How?**
- heapify()

```
if __name__ == '__main__':
    bh = BinHeap(False)
    bh.buildHeap([17, 15, 16, 12, 13, 99])
    bh.draw()

    bh.heapify(True)
    bh.draw()

    bh.heapify()
    bh.draw()

    bh.heapify(False)
    bh.draw()
```

```
         99
        / \
       /   \
      /     \
     15      17
    / \     /
   /   \   16
  12   13
```

max-heap

```
         12
        / \
       /   \
      /     \
     13      16
    / \     /
   /   \   17
  15   99
```

min-heap

```
         12
        / \
       /   \
      /     \
     13      16
    / \     /
   /   \   17
  15   99
```

no change
min-heap

```
         99
        / \
       /   \
      /     \
     15      17
    / \     /
   /   \   16
  12   13
```

max-heap

Heap sort

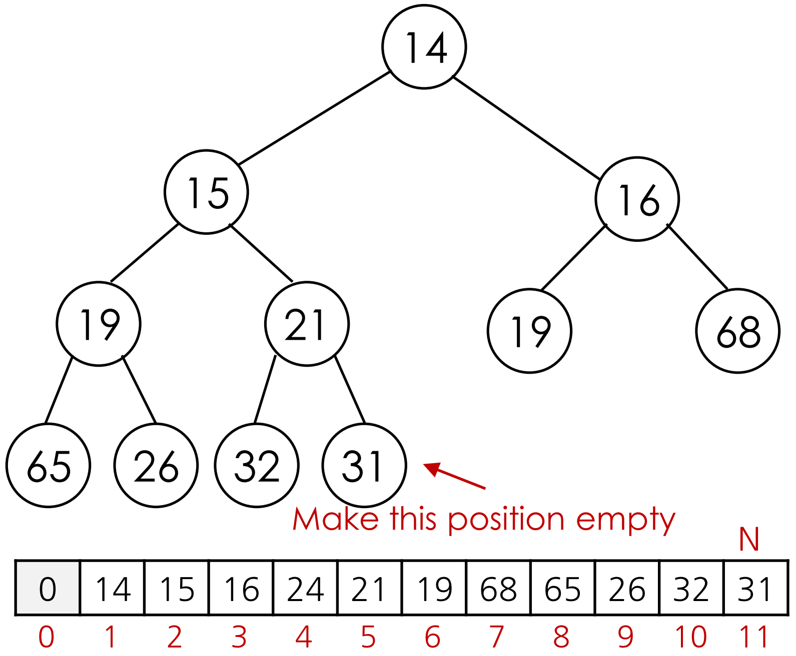
Algorithm:

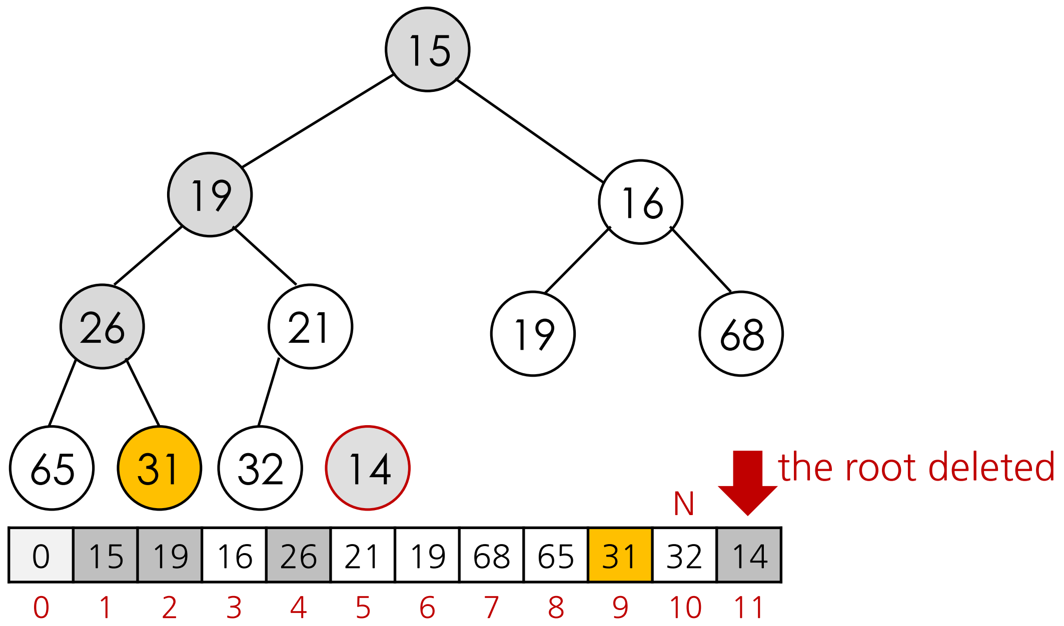
- Step 1: **heapify**(True/False)
  - min-heap returns in ascending order, max-heap in descending order in Step 2.
- Step 2: **delete the root** repeatedly
  - Collect all return values from this root delete operations,
    then, it is in ascending order if min-heap, descending order if max-heap
  - All the deleted roots saved at the reserved area of the heap are also sorted as well.
- Through this process of heap sort,
  we can get the sequence in both ascending and descending order at the same time.

Heap sort: delete() or dequeue()

```
class BinHeap:
    ...
    def delete(self):
        retval = self.heap[1]                  # root is saved to return
        self.heap[1] = self.heap[self.N]       # last element becomes root - need sink it
        self.N -= 1                            # reduce size by one
        self.heap.pop()                        # remove the last element (it will be unnecessary)
        self.sink(1)                           # now, sink down the root to make it heap-ordered
        return retval
```

- Do not discard the root deleted, but save the deleted root after index N slot.





Heap sort: delete() or dequeue()

```
class BinHeap:
    ...
    def delete(self):
        retval = self.heap[1]                  # root is saved to return
        self.heap[1] = self.heap[self.N]       # last element becomes root - need sink it
        self.N -= 1                            # reduce size by one
        self.heap.pop()                        # remove the last element (it will be unnecessary)
        self.sink(1)                           # now, sink down the root to make it heap-ordered
        return retval
```

```
class BinHeap:
    ...
    def insert(self, key):                 # check N and len(heap) before using
        self.heap.append(key)              # append() if necessary, otherwise use list index
        self.N += 1
        self.swim(self.N)
```

- Check whether or not there are some space after the index N.
- If there is no room, then use **`append(key),`**
  otherwise use **`self.heap[self.N] = key`** to utilize the room available,

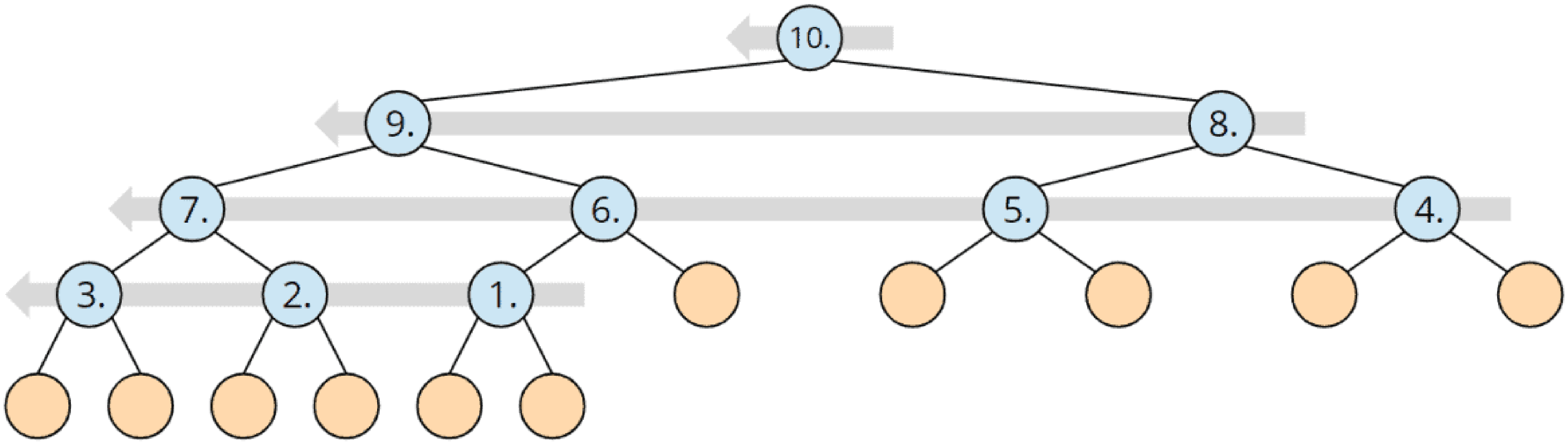
Heap sort: delete() or dequeue()

```python
if __name__ == '__main__':
    bh = BinHeap()
    bh.buildHeap([17, 15, 16, 12, 13, 99])
    print('N:', bh.N)
    print('len:', len(bh.heap))
    print('heap:', bh.heap)
    bh.draw()

    bh.insert(1)
    bh.insert(100)

    bh_sorted = [bh.delete() for x in range(bh.N)]
    print('sorted(by root):', bh_sorted)
    print('sorted(by save):', bh.heap[1:])
    print('N:', bh.N)
    print('len:', len(bh.heap))
    print('heap:', bh.heap)
    print('recover the original heap')
    bh.heapify()
    bh.draw()
```

```
N: 6
len: 7
heap: [0, 12, 13, 16, 15, 17, 99]
      12
     / \
    /   \
   /     \
  13      16
 / \      /
/   \   99
15   17
sorted(by root): [1, 12, 13, 15, 16, 17, 99, 100]
sorted(by save): [100, 99, 17, 16, 15, 13, 12, 1]
N: 0
len: 9
heap: [0, 100, 99, 17, 16, 15, 13, 12, 1]
recover the original heap
           1
          / \
         /   \
        /     \
       /       \
      /         \
    15           12
   / \          / \
  /   \        /   \
 16   100     13    17
 /
99
```

Time complexity: buildHeap() or heapify()

- To initially build the heap, buildHeap() process calls sink() for **each parent node (or the last internal node)** - backward, starting with the last node and ending at the tree root. A heap of size *n* has *n/2* (rounded down) parent nodes:



- The time complexity of the building heap is known as $O(n)$ and heap sort as $O(n \log n)$
- For proof, refer to here.

Summary

- The heap sort process has the two stages:
  - $O(n)$ time for buildHeap and $O(n \log n)$ to remove each node in order.
    Therefore, the time complexity of the heap sort is O(n log n).

- **BuildHeap():** O(N)
- **Heapsort():** O(N log N)

- **Proof:**
  - https://stackoverflow.com/questions/9755721/how-can-building-a-heap-be-on-time-complexity
  - https://www.insertingwiththeweb.com/data-structures/binary-heap/build-heap-proof/
  - https://www.quora.com/How-is-the-time-complexity-of-building-a-heap-is-o-n
- **References in Korean:**
  - https://ratsgo.github.io/data%20structure&algorithm/2017/09/27/heapsort/
  - https://zeddios.tistory.com/56

Data Structures in Python

- Heap and Priority Queue
- Heap Coding
- **Heap Sort & Min/MaxHeap**