

빅 데이터 혁신 공유 대학

파이썬으로 배우는 데이터 구조

한동대학교 전산전자공학부

김영섭 교수



교육부



한국연구재단



Data Structures in Python

Chapter 8

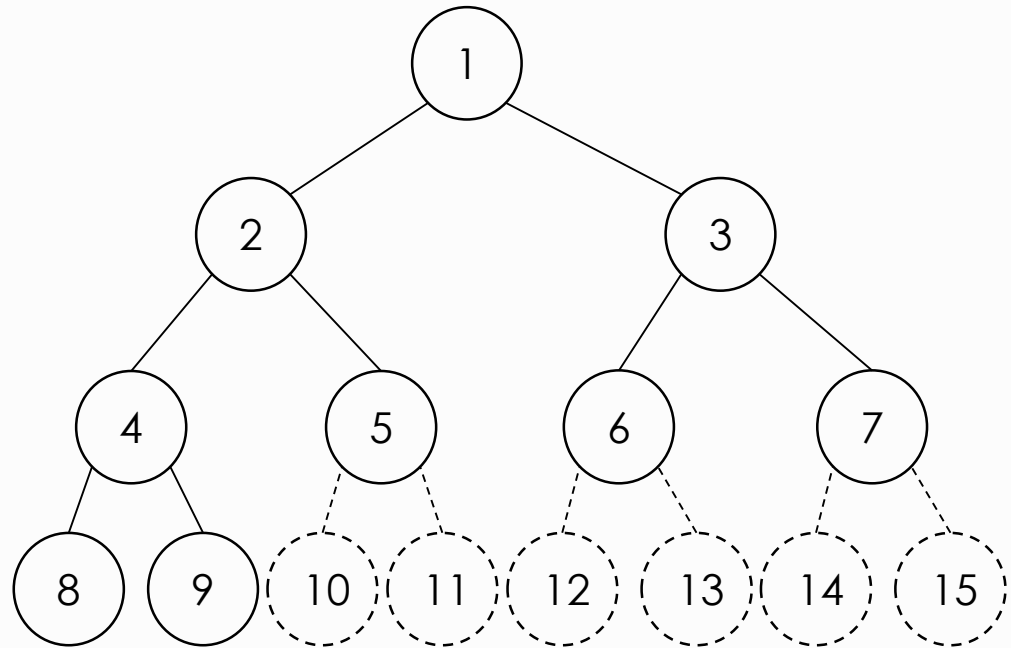
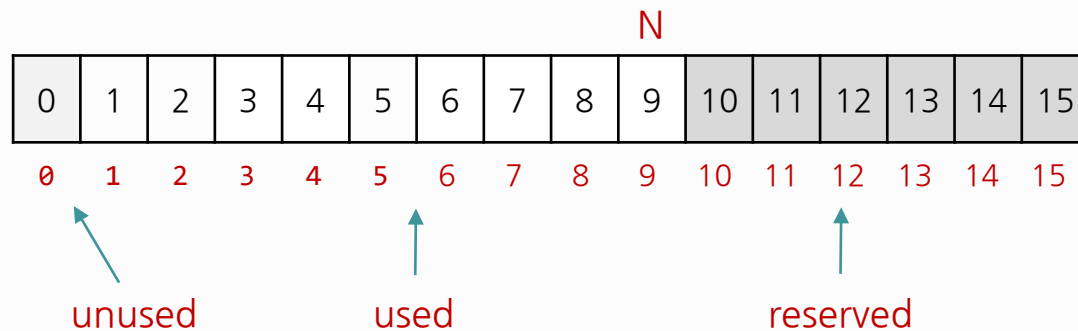
- Heap and Priority Queue
- Heap Coding
- **Heap Sort & Min/MaxHeap**

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 , $1 \leq i \leq n$, we have
 - $\text{parent}(i)$ is at $\lfloor i/2 \rfloor$ If $i = 1$, i is at the root and has no parent
 - $\text{leftChild}(i)$ is at $2i$ if $2i \leq n$. If $2i > n$, then i has no left child.
 - $\text{rightChild}(i)$ is at $2i+1$ if $2i+1 \leq n$. If $2i+1 > n$, then i has no right child.



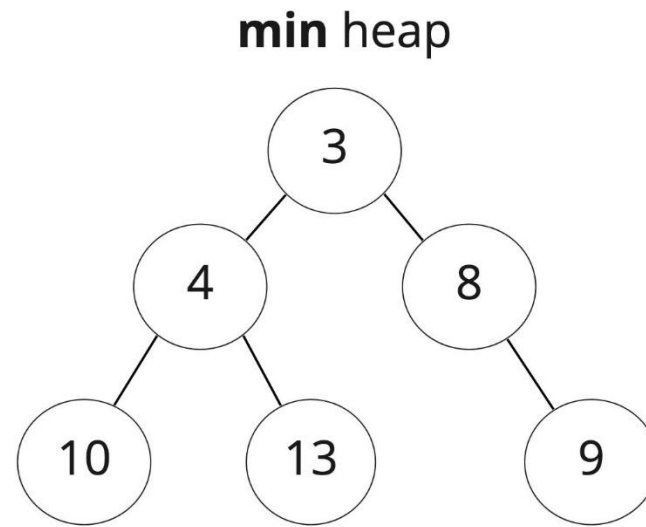
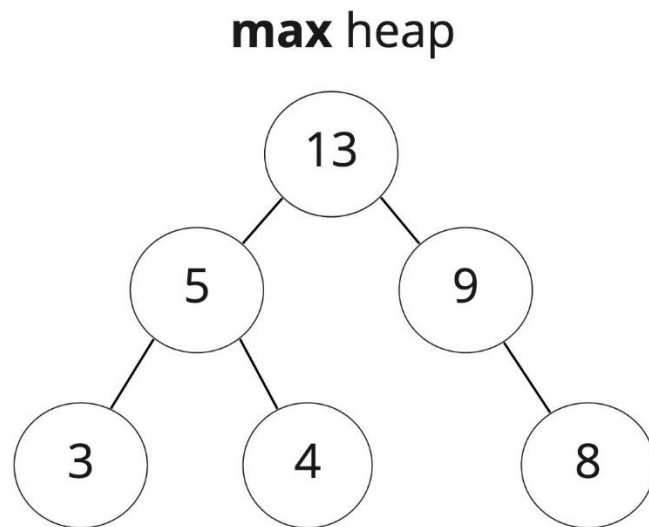
A complete binary tree

Min/Max-Heap Conversion

Algorithm

- Change the comparator
- heapify()

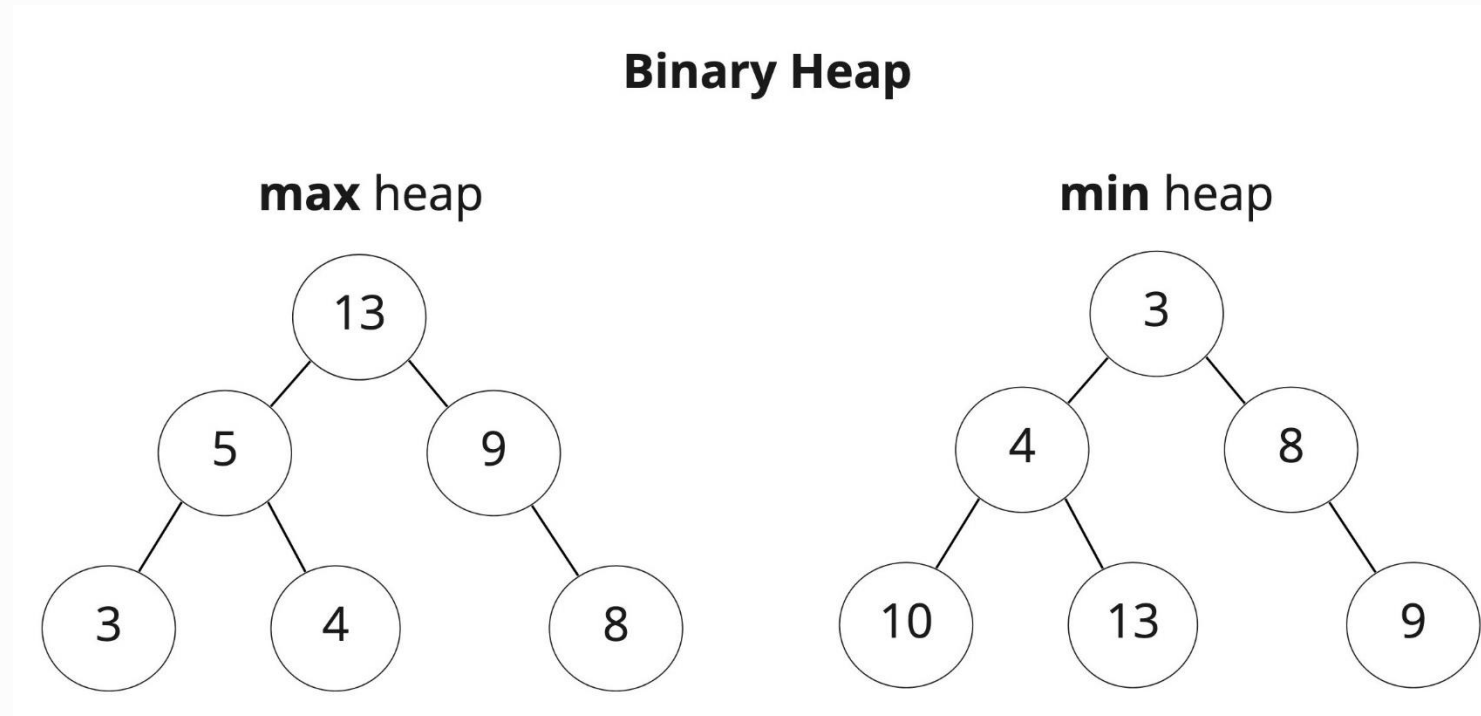
Binary Heap



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**

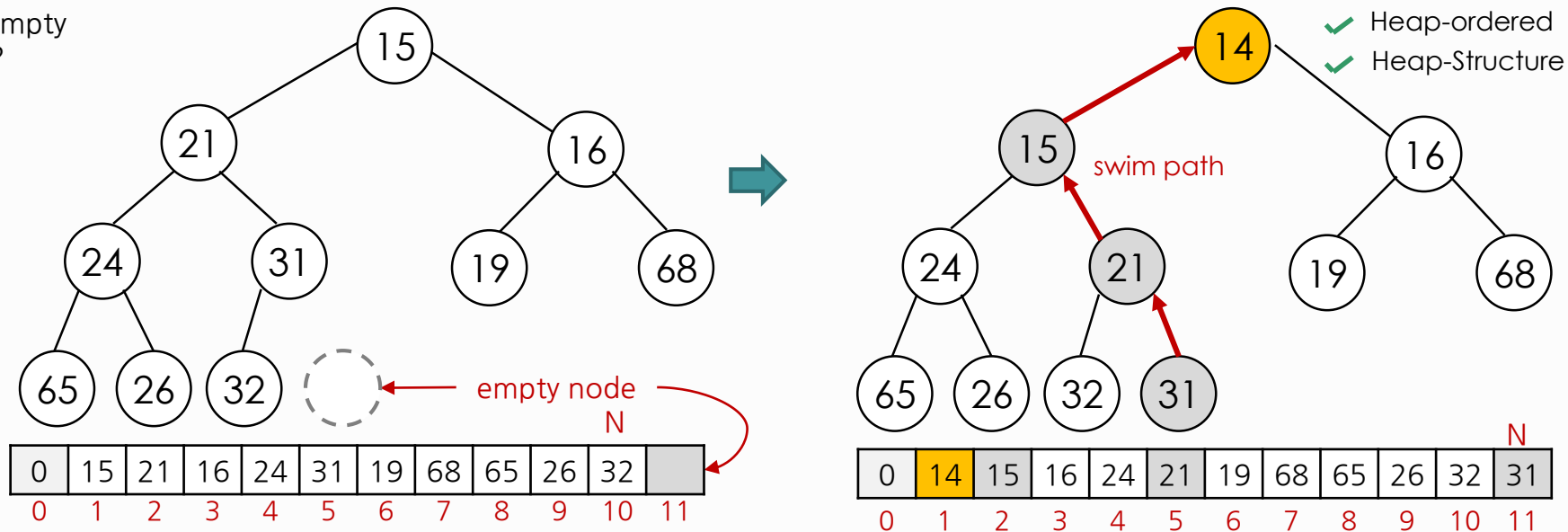
```
class BinHeap:
```

```
...
```

```
def insert(self, key):  
    self.heap.append(key)  
    self.N += 1  
    self.swim(self.N)
```

```
# check N and len(heap) before using  
# append() if necessary, otherwise use list index
```

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):
```

```
    while k // 2 > 0:
```

```
        if self.heap[k//2] > self.heap[k]:
```

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

```
            k = k // 2
```

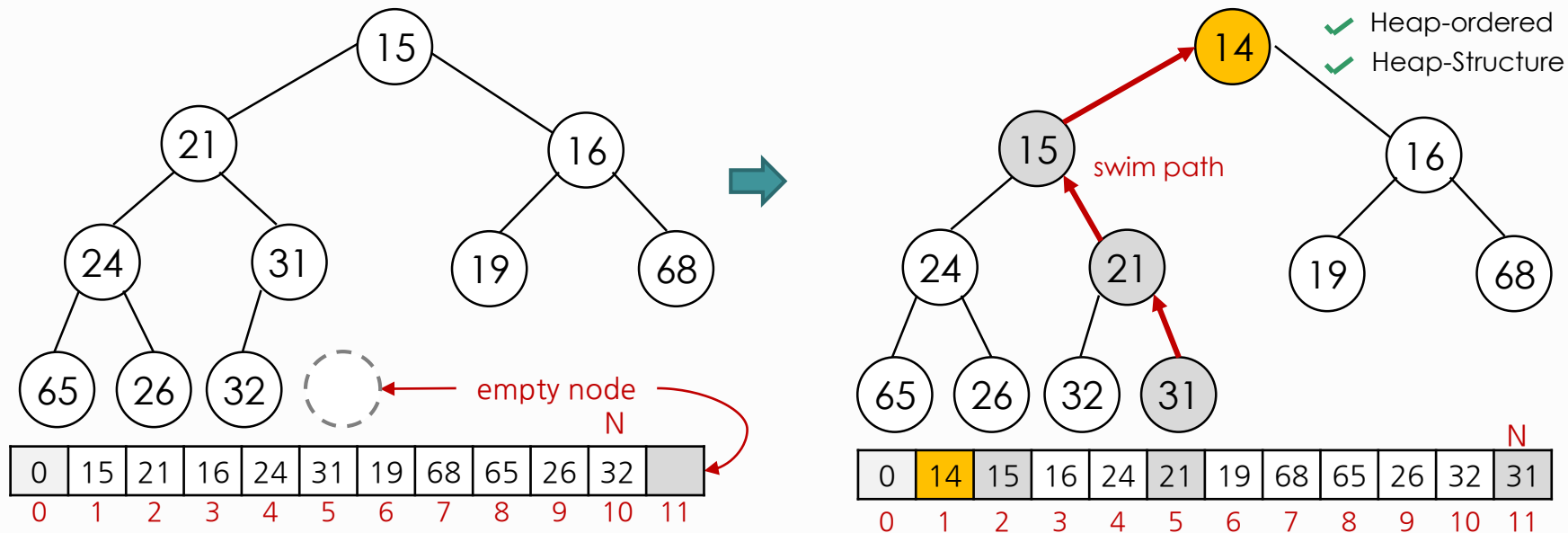
```
    # append key and swim up
```

```
    # if not reached root
```

```
    # if parent is more than kid (minheap)
```

```
    # swap(parent, kid)
```

```
    # 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):
        while k // 2 > 0:
            if self.heap[k//2] > self.heap[k]:
                self.swap(k//2, k)
            k = k // 2
```

append key and swim up
if not reached root
if parent is more than kid (minheap)
swap(parent, kid)
swim up - move to the parent node

```
class BinHeap:
    ...
    def sink(self, i):
        while (i * 2) <= self.N:
            k = 2 * i
            if k < self.N and self.heap[k] > self.heap[k+1]:
                k += 1
            if not self.heap[i] > self.heap[k]: break
            self.swap(i, k)
            i = k
```

start sink at node i
not bottom of tree yet?
left child
select one of two kids to compare
right child is selected
break if node i and kid are heap-ordered
if not heap-ordered, swap i and 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):
        while k // 2 > 0:
            if self.heap[k//2] > self.heap[k]:
                self.swap(k//2, k)
            k = k // 2
```

append key and swim up
if not reached root
if parent is more than kid (minheap)
swap(parent, kid)
swim up - move to the parent node

```
class BinHeap:
    ...
    def sink(self, i):
        while (i * 2) <= self.N:
            k = 2 * i
            if k < self.N and self.heap[k] > self.heap[k+1]:
                k += 1
            if not self.heap[i] > self.heap[k]: break
            self.swap(i, k)
            i = k
```

start sink at node i
not bottom of tree yet?
left child
select one of two kids to compare
right child is selected
break if node i and kid are heap-ordered
if not heap-ordered, swap i and 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):
        while k // 2 > 0:
            if self.heap[k//2] > self.heap[k]:
                self.swap(k//2, k)
            k = k // 2
```

append key and swim up
if not reached root
if parent is more than kid (minheap)
swap(parent, kid)
swim up - move to the parent node

```
class BinHeap:
    ...
    def sink(self, i):
        while (i * 2) <= self.N:
            k = 2 * i
            if k < self.N and self.heap[k] > self.heap[k+1]:
                k += 1
            if not self.heap[i] > self.heap[k]: break
            self.swap(i, k)
            i = k
```

start sink at node i
not bottom of tree yet?
left child
select one of two kids to compare
right child is selected
break if node i and kid are heap-ordered
if not heap-ordered, swap i and 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):
```

```
    while k // 2 > 0:
```

```
        if self.heap[k//2] > self.heap[k]:
```

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

```
        k = k // 2
```

```
    # append key and swim up
```

```
    # if not reached root
```

```
    # if parent is more than kid (minheap)
```

```
    # swap(parent, kid)
```

```
    # swim up - move to the parent node
```

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

```
class BinHeap:
```

```
...
```

```
def sink(self, i):
```

```
    while (i * 2) <= self.N:
```

```
        k = 2 * i
```

```
        if k < self.N and self.heap[k] > self.heap[k+1]:
```

```
            k += 1
```

```
        if not self.heap[i] > self.heap[k]: break
```

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

```
        i = k
```

```
    # start sink at node i
```

```
    # not bottom of tree yet?
```

```
    # left child
```

```
    # select one of two kids to compare
```

```
    # right child is selected
```

```
    # break if node i and kid are heap-ordered
```

```
    # if not heap-ordered, swap i and 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):
        self.heap = [0]
        self.N = 0
        self.comp = None
        # min-heap by default
        # list, index 0 is not used
        # points the last valid heap element index
        # used later, comparator to switch between min-heap & max-heap
```

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

```
    while k // 2 > 0:
```

```
        if self.heap[k//2] > self.heap[k]:
```

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

```
        k = k // 2
```

```
    # append key and swim up
```

```
    # if not reached root
```

```
    # if parent is more than kid (minheap)
```

```
    # swap(parent, kid)
```

```
    # swim up - move to the parent node
```

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

```
class BinHeap:
```

```
...
```

```
def sink(self, i):
```

```
    while (i * 2) <= self.N:
```

```
        k = 2 * i
```

```
        if k < self.N and self.heap[k] > self.heap[k+1]: # select one of two kids to compare
```

```
            k += 1
```

```
        if not self.heap[i] > self.heap[k]: break # break if node i and kid are heap-ordered
```

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

```
        i = k
```

```
    # start sink at node i
```

```
    # not bottom of tree yet?
```

```
    # left child
```

```
    # right child is selected
```

```
    # if not heap-ordered, swap i and 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

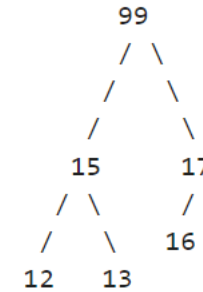
def heapify(self, min = None):
    self.buildHeap(self.heap[1:], min)
```

Min/Max-Heap Conversion

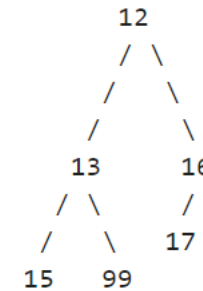
Algorithm

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

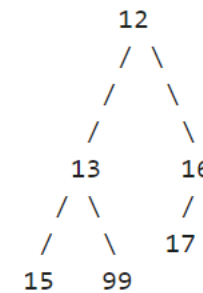
```
if __name__ == '__main__':  
    bh = BinHeap(False)  
    bh.buildHeap([17, 15, 16, 12, 13, 99])  
    bh.draw()  
  
    bh.heapify(True)  
    bh.draw()  
  
    bh.heapify()  
    bh.draw()  
  
    bh.heapify(False)  
    bh.draw()
```



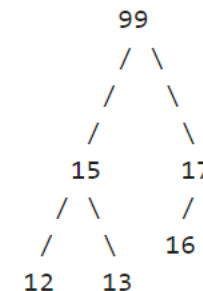
max-heap



min-heap



no change
min-heap



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]
```

```
    self.heap[1] = self.heap[self.N]
```

```
    self.N -= 1
```

```
    self.heap.pop()
```

```
    self.sink(1)
```

```
    return retval
```

```
# root is saved to return
```

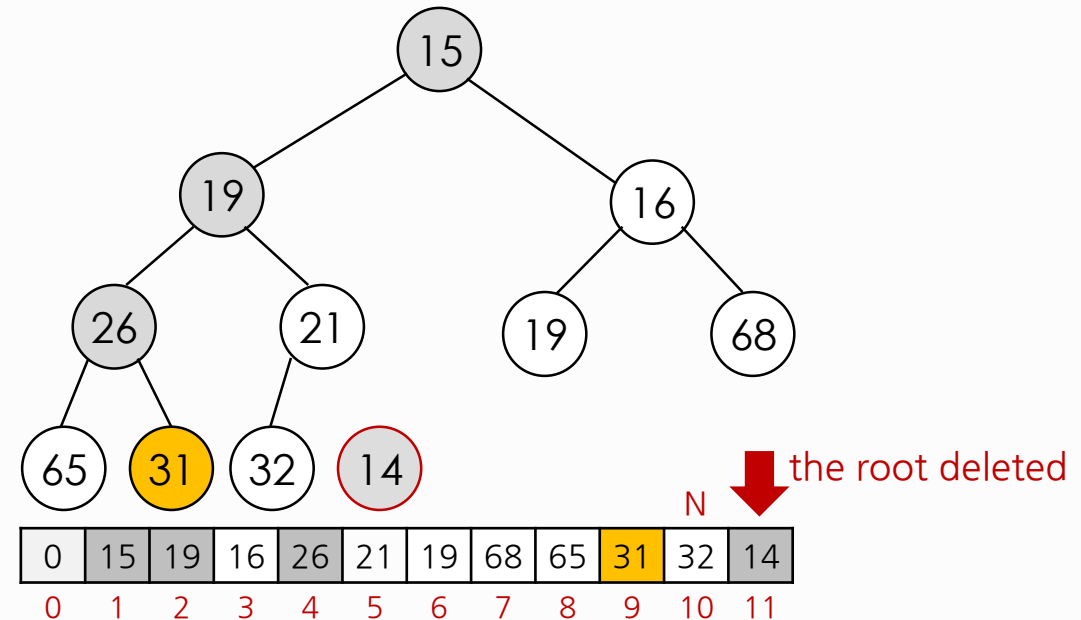
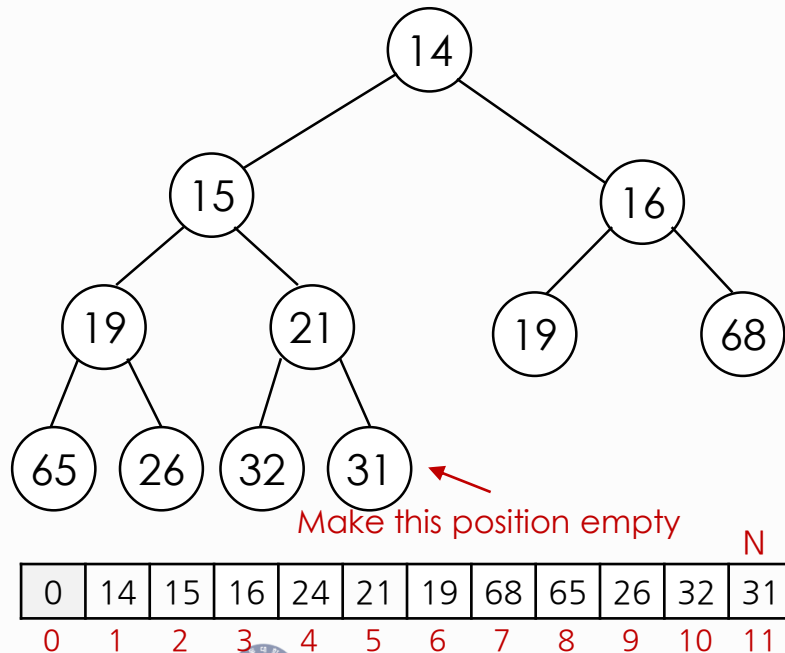
```
# last element becomes root - need sink it
```

```
# reduce size by one
```

```
# remove the last element (it will be unnecessary)
```

```
# now, sink down the root to make it heap-ordered
```

- 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):
        self.heap.append(key)                # check N and len(heap) before using
        self.N += 1                          # append() if necessary, otherwise use list index
        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,

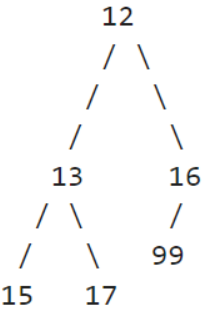
Heap sort: delete() or dequeue()

```
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()

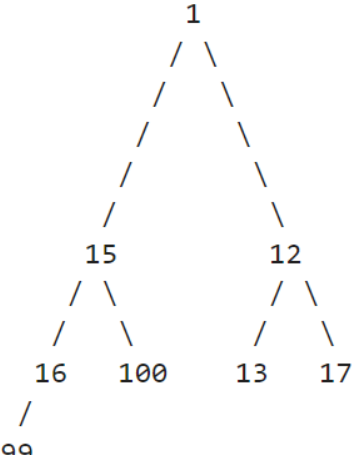
    bh.insert(1)
    bh.insert(100)

    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]
```

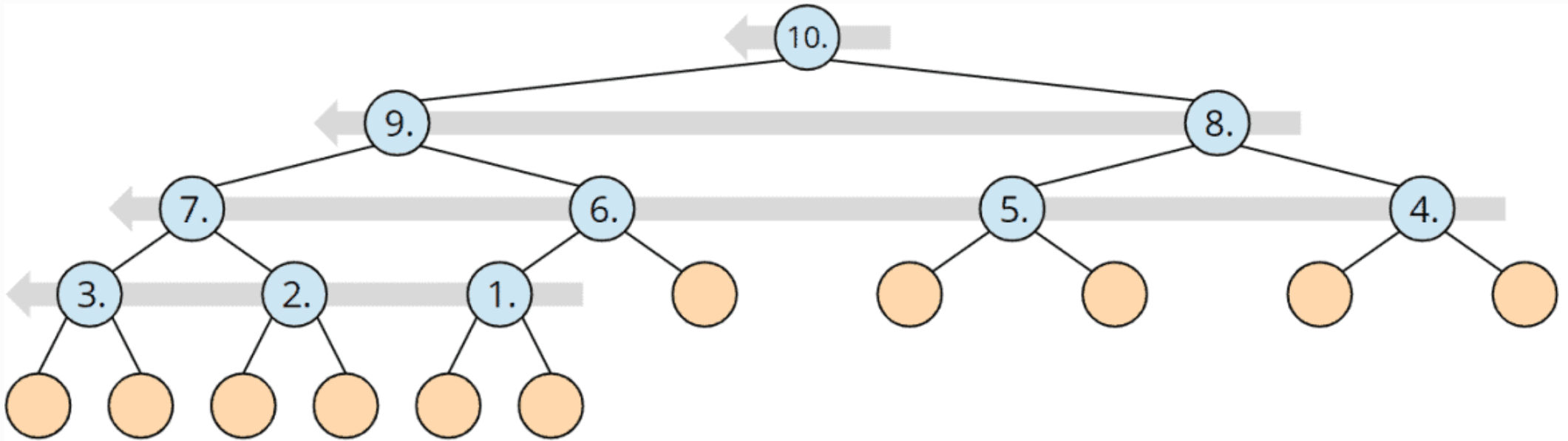


```
sorted(by root): [1, 12, 13, 15, 16, 17, 99, 100]
sorted(by save): [100, 99, 17, 16, 15, 13, 12, 1]
N: 0
len: 9
heap: [0, 100, 99, 17, 16, 15, 13, 12, 1]
recover the original heap
```



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

Chapter 8

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