

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**
- Min/MaxHeap and Heap sort

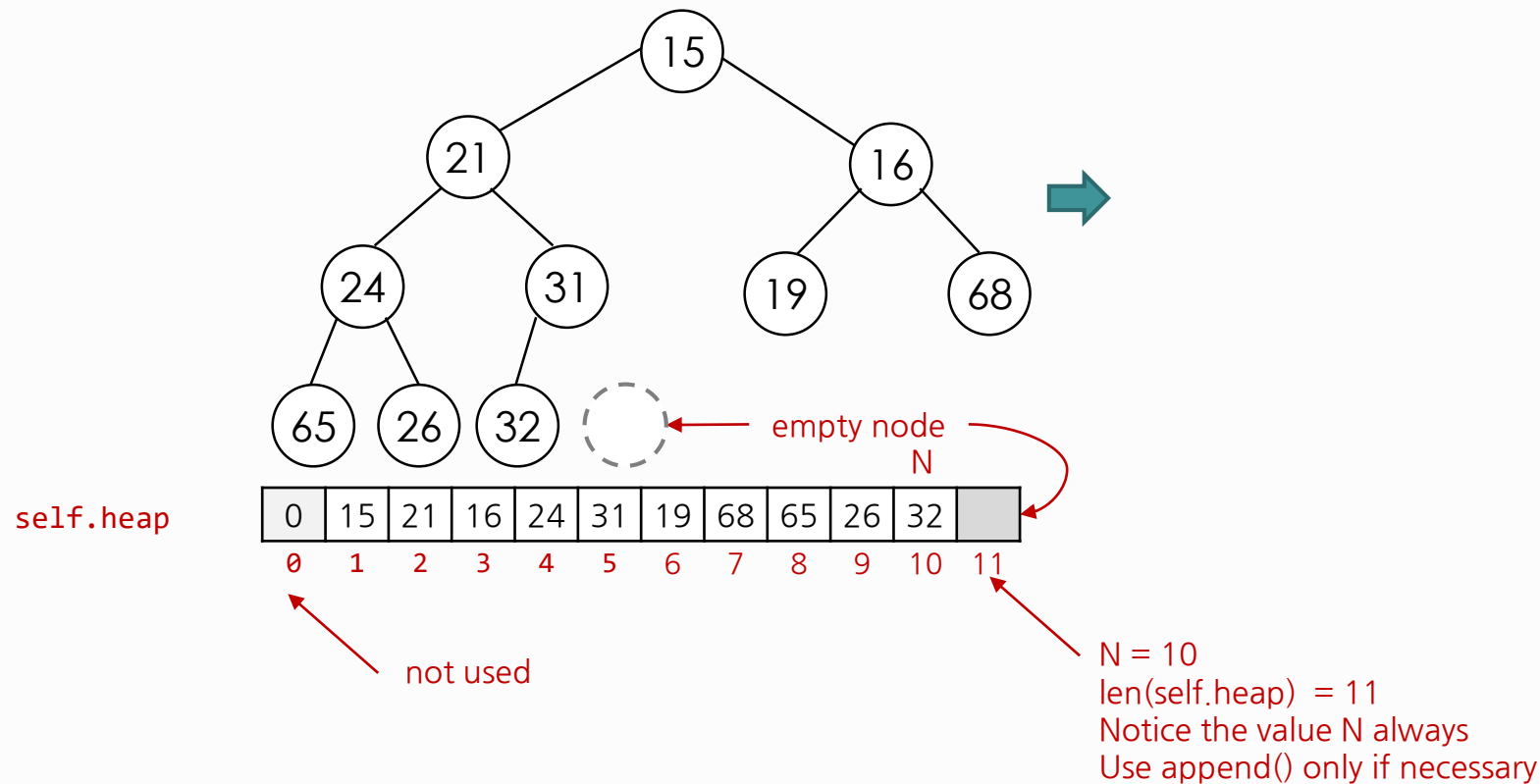
Agenda & Readings

- Heap and Priority Queue
 - Heap Class and Constructor
 - Heap ADT:
 - Insert(), Delete()
 - HeapBuild(), Heapify()
 - Helper functions - swim(), swap(), sink()
- Reference:
 - Problem Solving with Algorithms and Data Structures

Heap Class and Constructor

```
class BinHeap:
    def __init__(self):
        self.heap = [0]
        self.N = 0
        self.comp = None
```

set more for min-heap, less for max-heap
list, index 0 is not used
points the last valid heap element index
used later, **comparator to switch between min-heap & max-heap**



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**

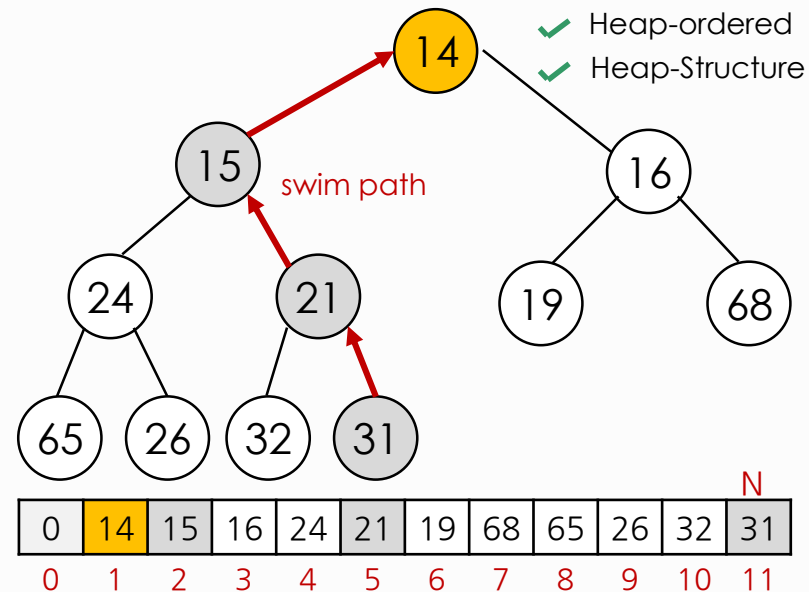
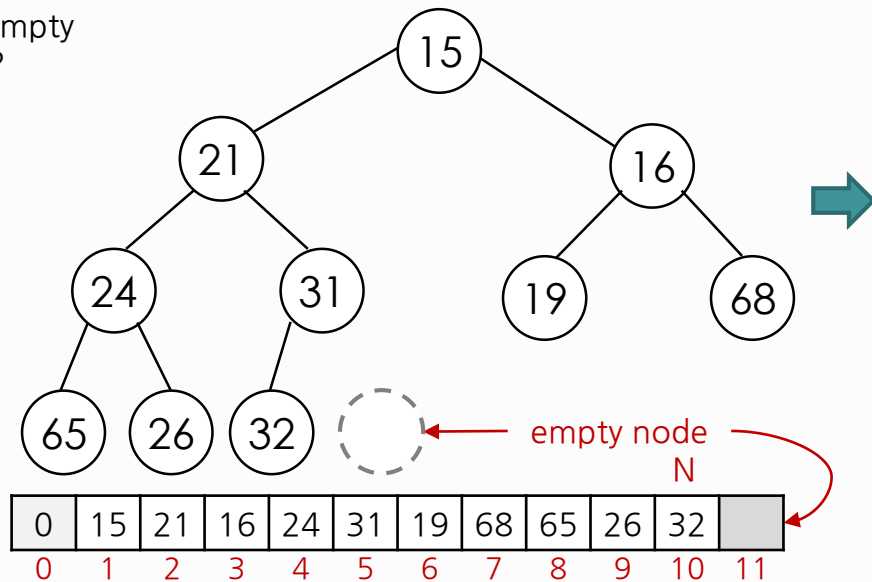
```
class BinHeap:
```

```
...
```

```
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
```

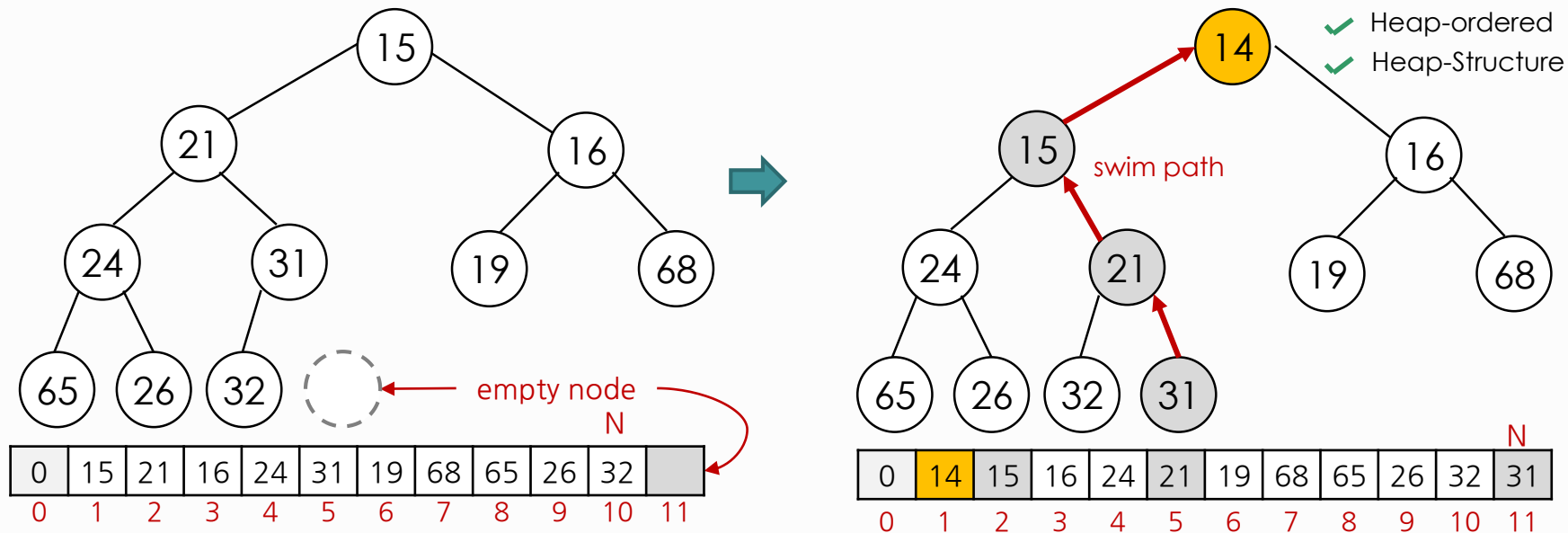
```
    # append key and swim up
```

```
    # if not reached root
```

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

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

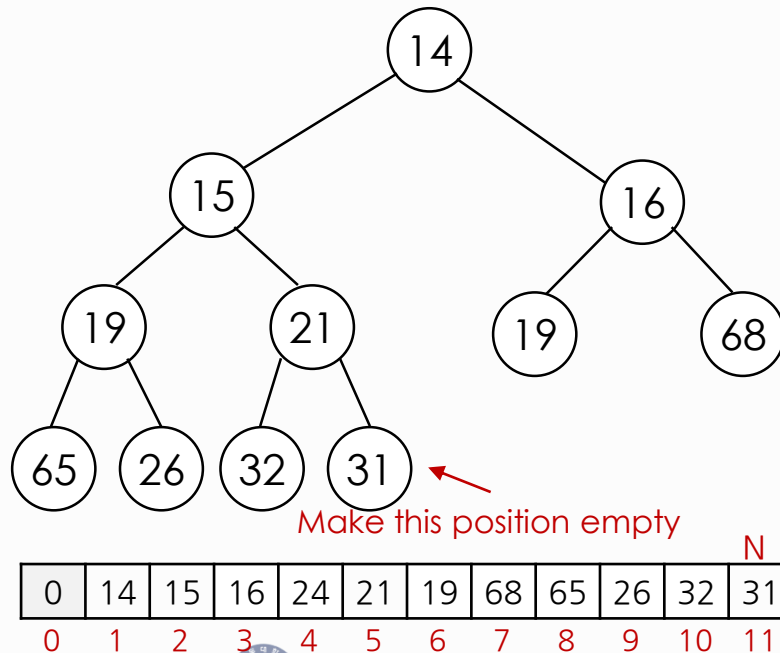
```
    # swim up - move to the parent node
```



min-heap: delete() or dequeue()

Algorithm:

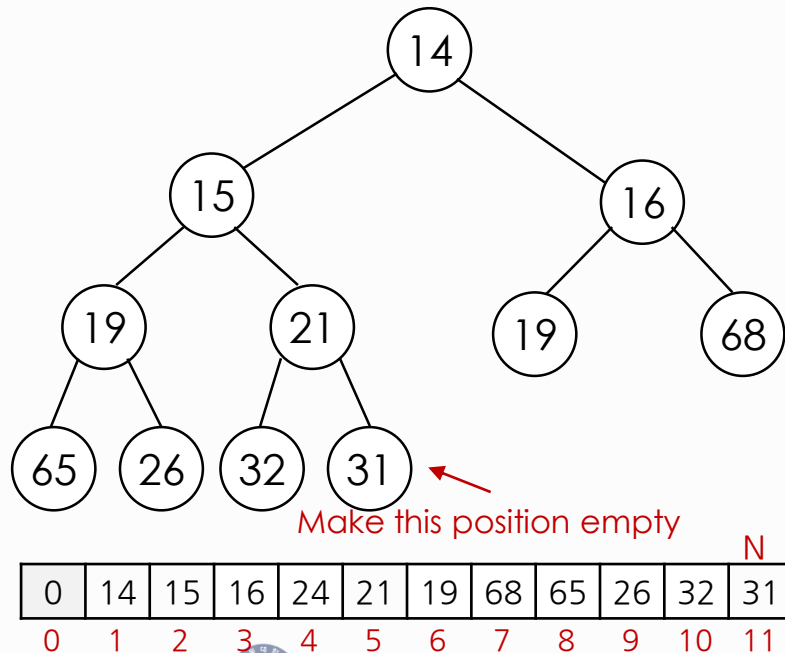
- Swap the root and the last element.
- Heap decreases by one in size.
- **Move down (sink) the root** while not satisfying heap-ordered.
 - Minimum element is **always** at the root (by min-heap definition).



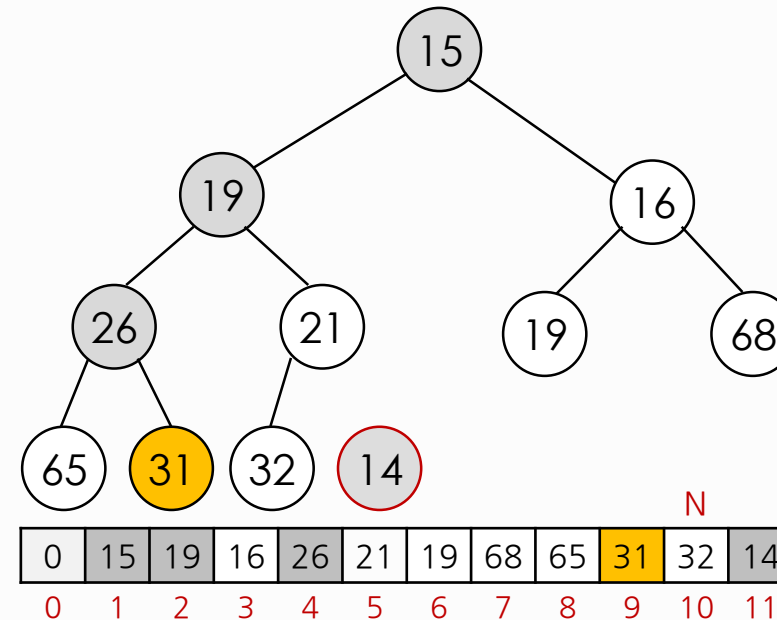
min-heap: delete() or dequeue()

Algorithm:

- Swap the root and the last element.
- Heap decreases by one in size.
- **Move down (sink) the root** while not satisfying heap-ordered.
 - Minimum element is **always** at the root (by min-heap definition).



- heap-ordered?
- **sink()**: select one of two children and compare



- ✓ Heap-ordered
- ✓ Heap-Structure

min-heap: 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
```

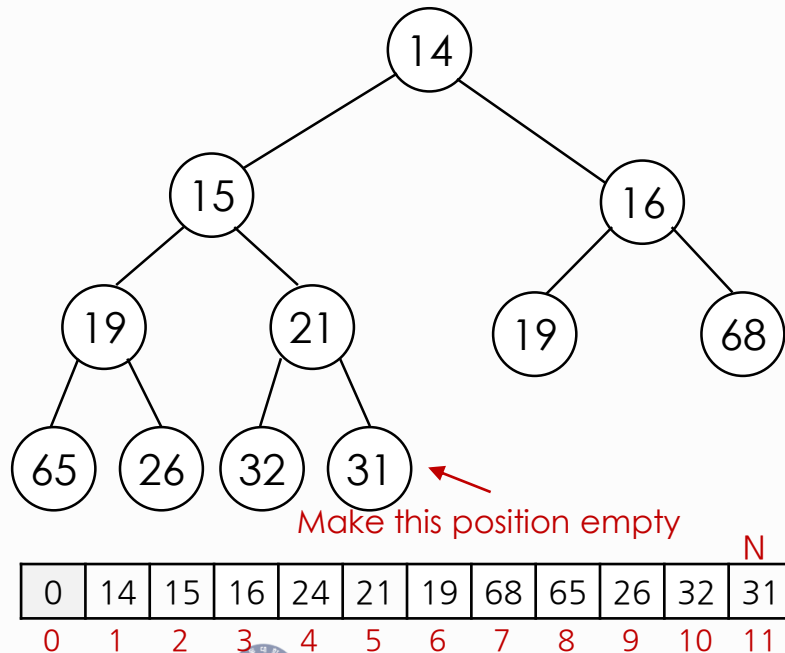
```
# root is saved to return
```

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

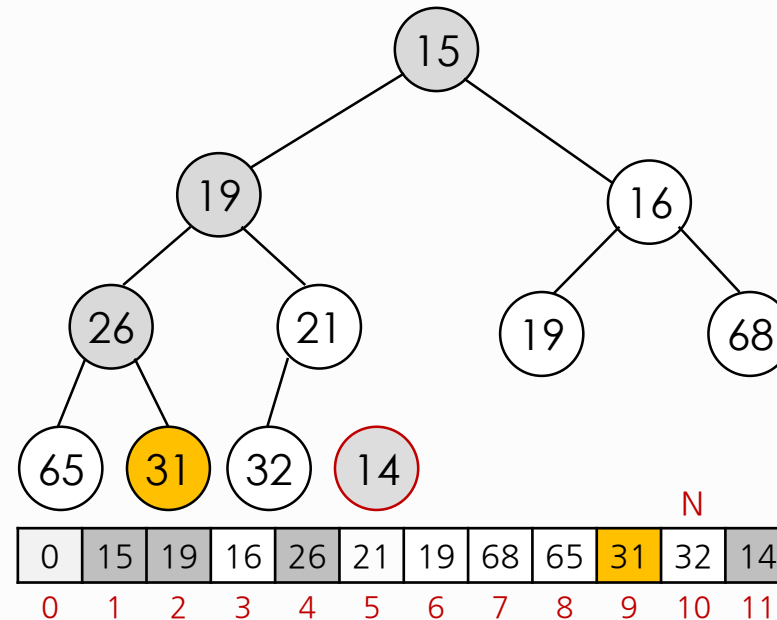
```
# reduce size by one
```

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

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



- heap-ordered?
- **sink()**: select one of two children and compare



- ✓ Heap-ordered
- ✓ Heap-Structure

min-heap: delete() or dequeue()

```
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
```

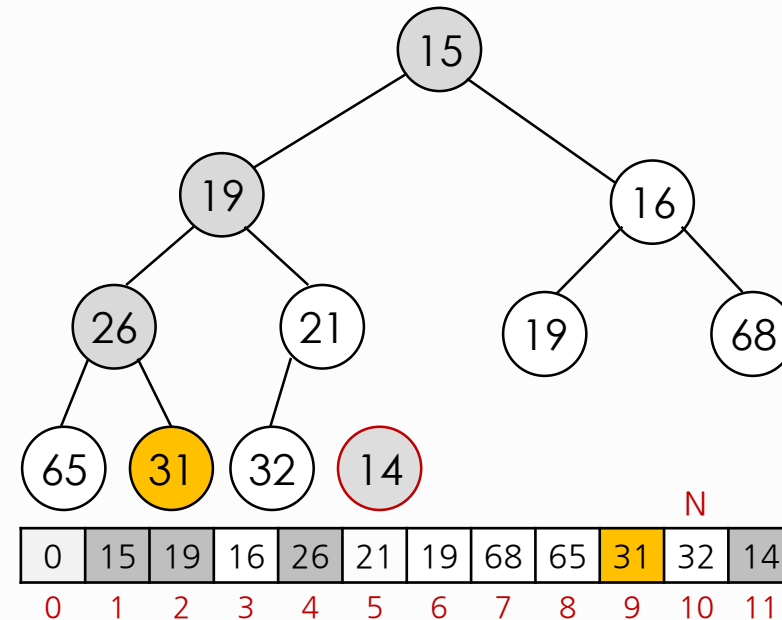
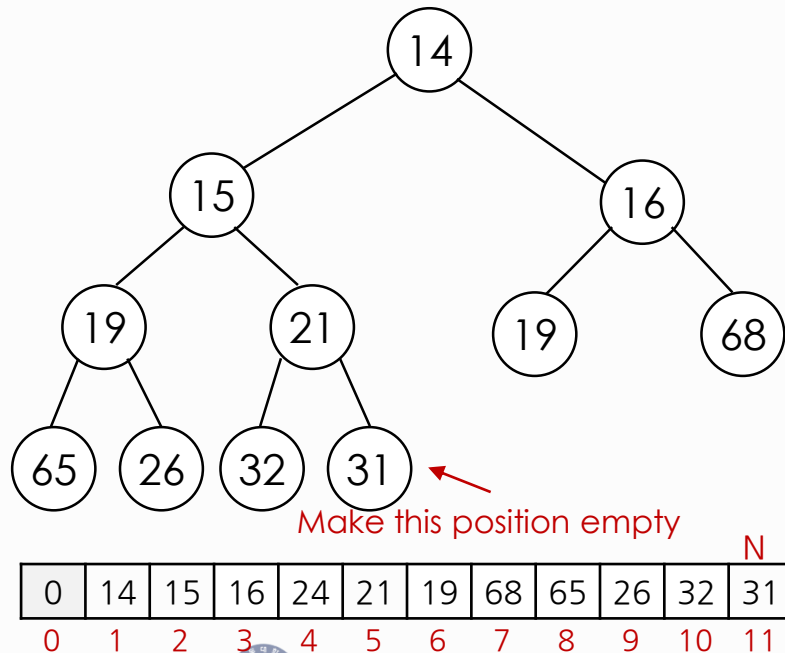
```
# not bottom of tree yet?
```

```
# left child
```

```
# right child is selected
```

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

```
# i becomes k & continue sink process
```

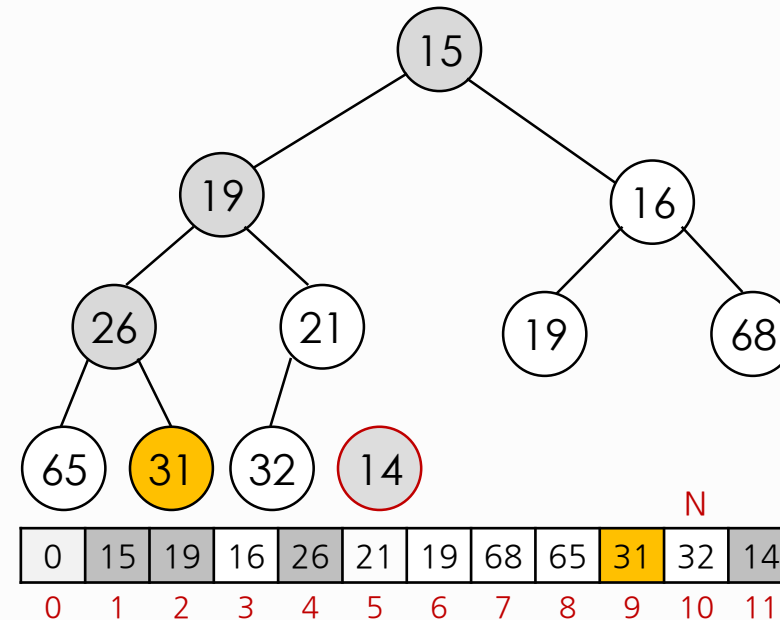
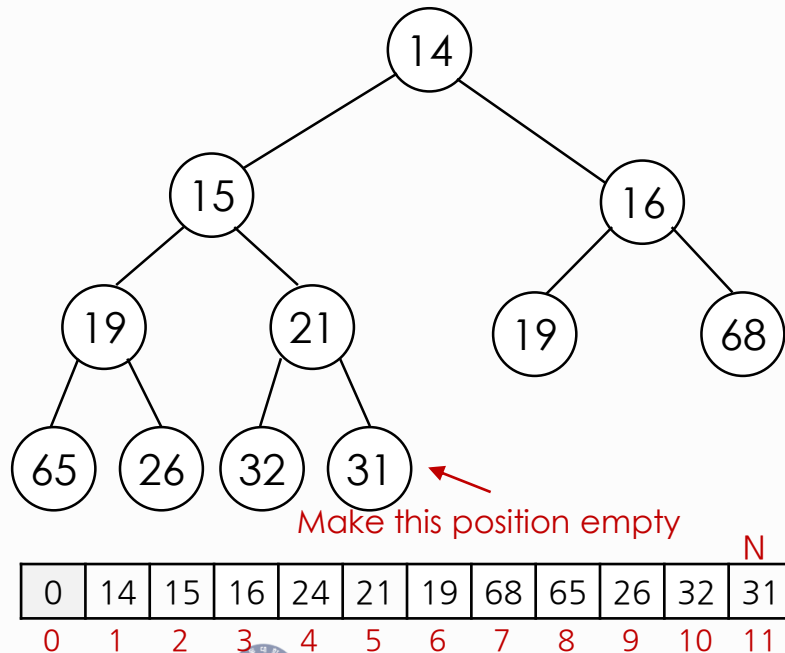


✓ Heap-ordered
✓ Heap-Structure

min-heap: delete() or dequeue()

- What do you expect from the following code snippet?

```
result = [ bh.delete() for x in range(bh.N) ]
print('      result:', result)
print('number of elements N:', bh.N)
print('  length of heap list:', len(bh.heap))
print('    heap list stored:', bh.heap)
```



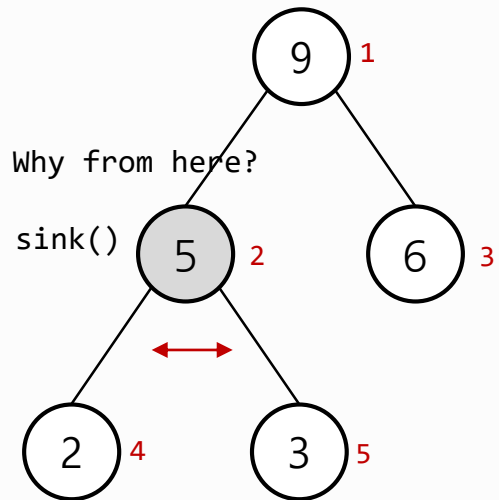
✓ Heap-ordered
✓ Heap-Structure

min-heap: buildHeap() and heapify()

```
class BinHeap:
    ...
    def buildHeap(self, arr):
        self.heap = [0] + arr[:]
        self.N = len(arr)
        i = len(arr) // 2
        while i > 0:
            self.sink(i)
            i -= 1
```

build heap from input arr list
set the initial heap
set the size
get the last internal node
sink from the last internal node to root 1

Initial heap



N=5

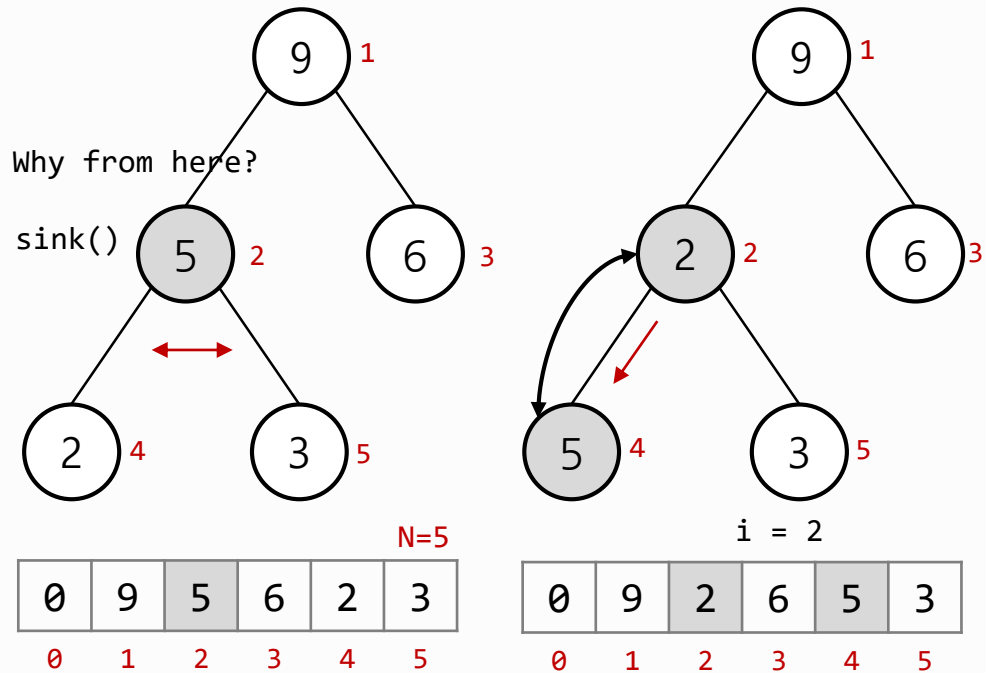
0	9	5	6	2	3
0	1	2	3	4	5

min-heap: buildHeap() and heapify()

```
class BinHeap:
    ...
    def buildHeap(self, arr):
        self.heap = [0] + arr[:]
        self.N = len(arr)
        i = len(arr) // 2
        while i > 0:
            self.sink(i)
            i -= 1
```

build heap from input arr list
set the initial heap
set the size
get the last internal node
sink from the last internal node to root 1

Initial heap



3

min-heap: buildHeap() and heapify()

```
class BinHeap:
```

```
...
```

```
def buildHeap(self, arr):
```

```
    self.heap = [0] + arr[:]
```

```
    self.N = len(arr)
```

```
    i = len(arr) // 2
```

```
    while i > 0:
```

```
        self.sink(i)
```

```
        i -= 1
```

```
# build heap from input arr list
```

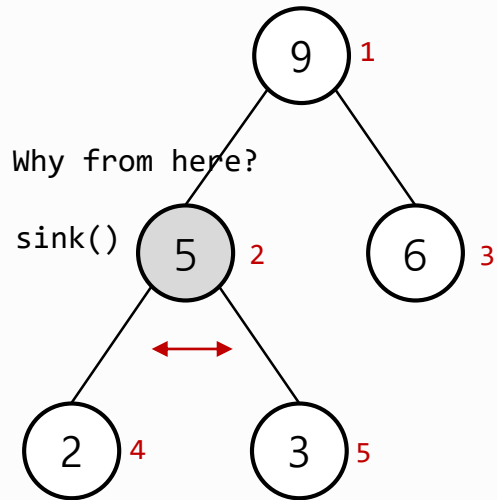
```
# set the initial heap
```

```
# set the size
```

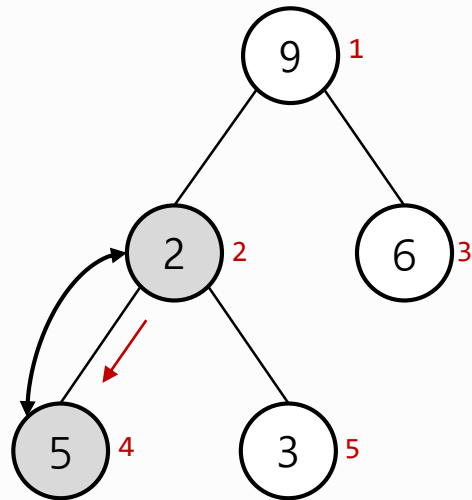
```
# get the last internal node
```

```
# sink from the last internal node to root 1
```

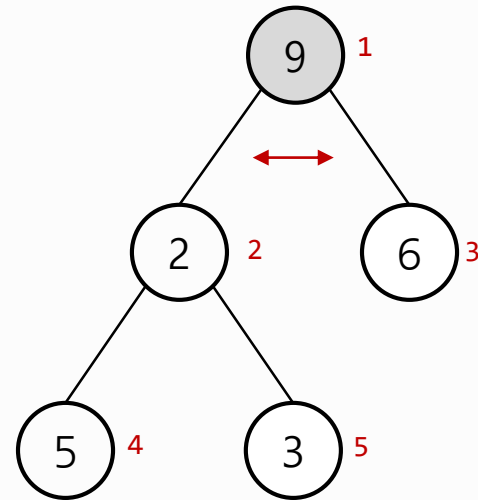
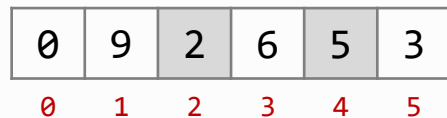
Initial heap



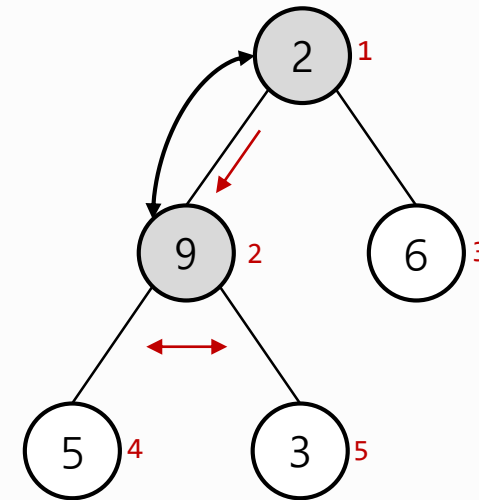
N=5



i = 2



i = 1



i = 1



min-heap: buildHeap() and heapify()

```
class BinHeap:
```

```
...
```

```
def buildHeap(self, arr):
```

```
    self.heap = [0] + arr[:]
```

```
    self.N = len(arr)
```

```
    i = len(arr) // 2
```

```
    while i > 0:
```

```
        self.sink(i)
```

```
        i -= 1
```

```
# build heap from input arr list
```

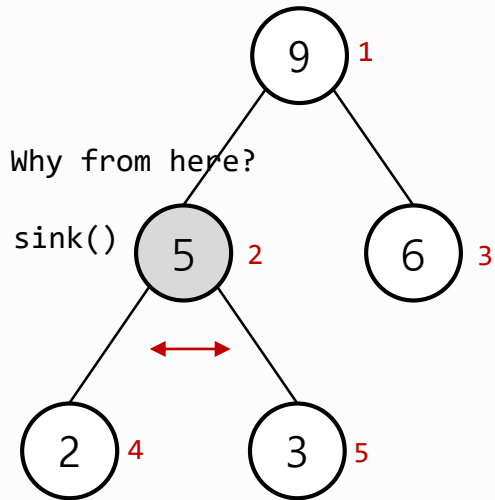
```
# set the initial heap
```

```
# set the size
```

```
# get the last internal node
```

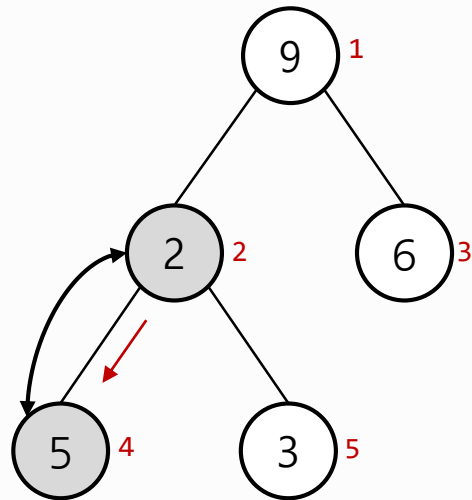
```
# sink from the last internal node to root 1
```

Initial heap



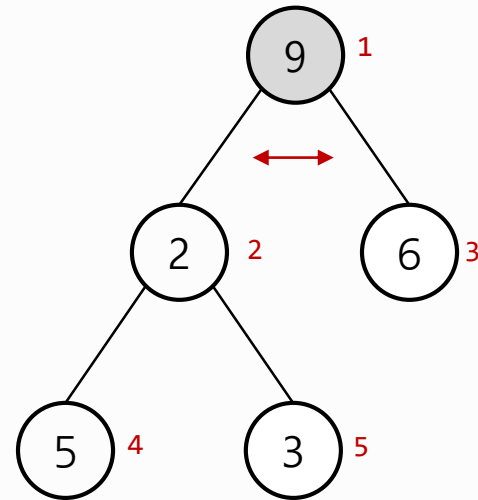
N=5

0	9	5	6	2	3
0	1	2	3	4	5



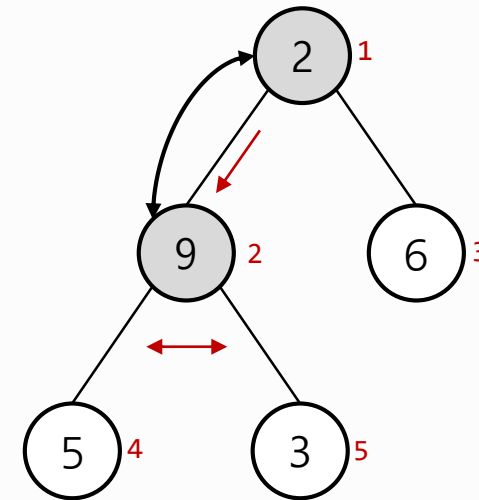
i = 2

0	9	2	6	5	3
0	1	2	3	4	5



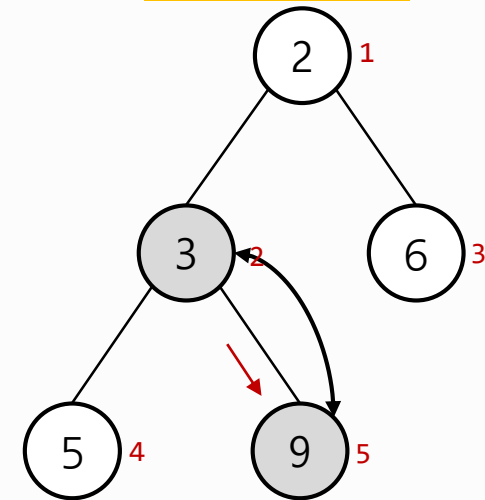
i = 1

0	9	2	6	5	3
0	1	2	3	4	5



i = 1

0	2	9	6	5	3
0	1	2	3	4	5

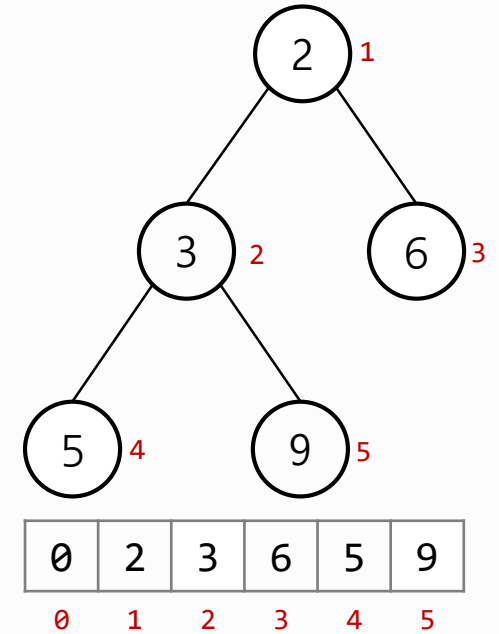


heap-ordered

0	2	3	6	5	9
0	1	2	3	4	5

min-heap: buildHeap() and heapify()

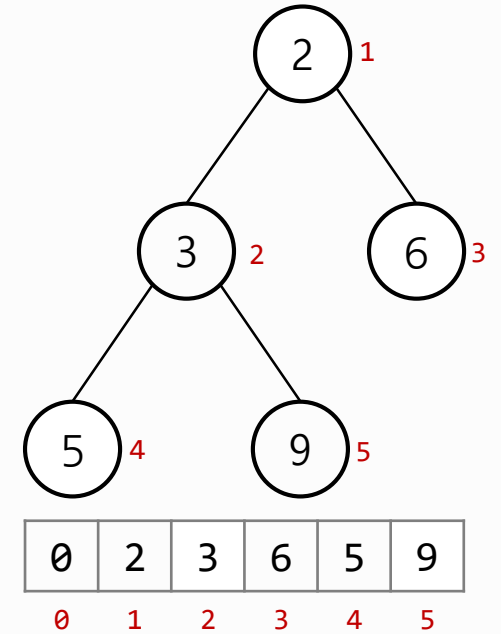
```
if __name__ == '__main__':  
    bh = BinHeap()  
    bh.buildHeap([9, 5, 6, 2, 3])  
    print('number of elements N:', bh.N)  
    print('  lengh of heap list:', len(bh.heap))  
    print('    heap list stored:', bh.heap)  
    bh.draw()  
  
    print('\ninserting: 7 - already heap-ordered')  
    bh.insert(7)
```



min-heap: buildHeap() and heapify()

```
if __name__ == '__main__':
    bh = BinHeap()
    bh.buildHeap([9, 5, 6, 2, 3])
    print('number of elements N:', bh.N)
    print('  length of heap list:', len(bh.heap))
    print('    heap list stored:', bh.heap)
    bh.draw()

    print('\ninserting: 7 - already heap-ordered')
    bh.insert(7)
    bh.draw()
    print('\ninserting: 1 - swim up, become root')
    bh.insert(1)
    bh.draw()
```



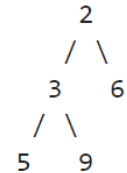
min-heap: buildHeap() and heapify()

```
if __name__ == '__main__':
    bh = BinHeap()
    bh.buildHeap([9, 5, 6, 2, 3])
    print('number of elements N:', bh.N)
    print('length of heap list:', len(bh.heap))
    print('heap list stored:', bh.heap)
    bh.draw()

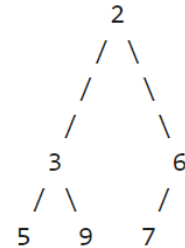
    print('\ninserting: 7 - already heap-ordered')
    bh.insert(7)
    bh.draw()
    print('\ninserting: 1 - swim up, become root')
    bh.insert(1)
    bh.draw()

    print('\ndeleting root to sort - heap depleted')
    bh_sorted = [ bh.delete() for x in range(bh.N) ]
    print('bh_sorted:', bh_sorted)
    print('number of elements N:', bh.N)
    print('length of heap list:', len(bh.heap))
    print('heap list stored:', bh.heap)
```

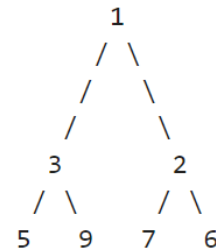
number of elements N: 5
length of heap list: 6
heap list stored: [0, 2, 3, 6, 5, 9]



inserting: 7 - already heap-ordered



inserting: 1 - swim up, become root



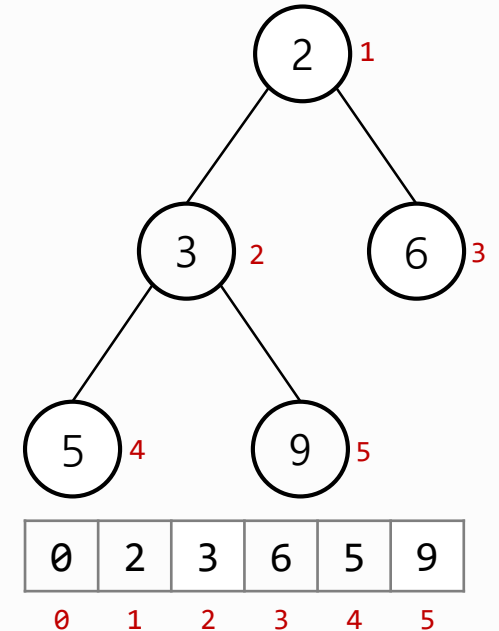
deleting root to sort - heap depleted

bh_sorted: [1, 2, 3, 5, 6, 7, 9]

number of elements N: 0

length of heap list: 1

heap list stored: [0]



heap elements are deleted
need to change not to delete
to have .

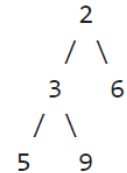
min-heap: buildHeap() and heapify()

```
if __name__ == '__main__':
    bh = BinHeap()
    bh.buildHeap([9, 5, 6, 2, 3])
    print('number of elements N:', bh.N)
    print('length of heap list:', len(bh.heap))
    print('heap list stored:', bh.heap)
    bh.draw()

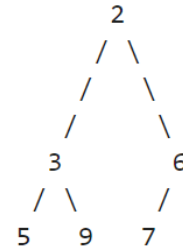
    print('\ninserting: 7 - already heap-ordered')
    bh.insert(7)
    bh.draw()
    print('\ninserting: 1 - swim up, become root')
    bh.insert(1)
    bh.draw()

    print('\ndeleting root to sort - heap depleted')
    bh_sorted = [ bh.delete() for x in range(bh.N) ]
    print('bh_sorted:', bh_sorted)
    print('number of elements N:', bh.N)
    print('length of heap list:', len(bh.heap))
    print('heap list stored:', bh.heap)
```

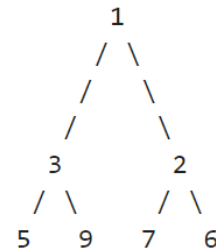
number of elements N: 5
length of heap list: 6
heap list stored: [0, 2, 3, 6, 5, 9]



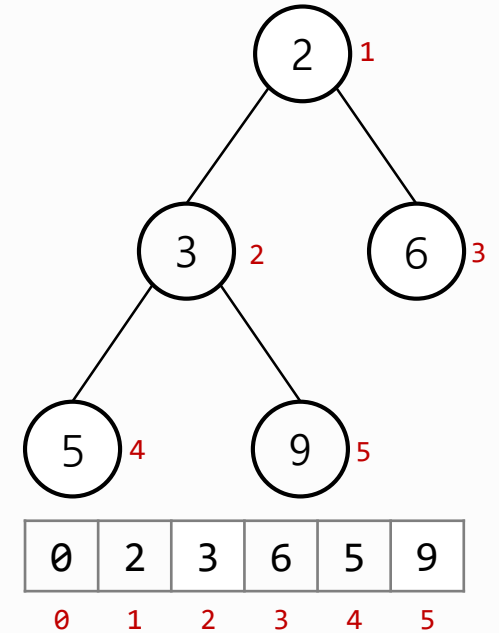
inserting: 7 - already heap-ordered



inserting: 1 - swim up, become root



deleting root to sort - heap depleted
bh_sorted: [1, 2, 3, 5, 6, 7, 9]
number of elements N: 0
length of heap list: 1
heap list stored: [0]



deleting root to sort - heap depleted
bh_sorted: [1, 2, 3, 5, 6, 7, 9]
number of elements N: 0
length of heap list: 8
heap list stored: [0, 9, 7, 6, 5, 3, 2, 1]

heap sort



heap elements are deleted
need to change not to delete
to have .

Data Structures in Python

Chapter 8

- Heap and Priority Queue
- **Heap Coding**
- Min/MaxHeap and Heap sort