Data Structures in Python

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

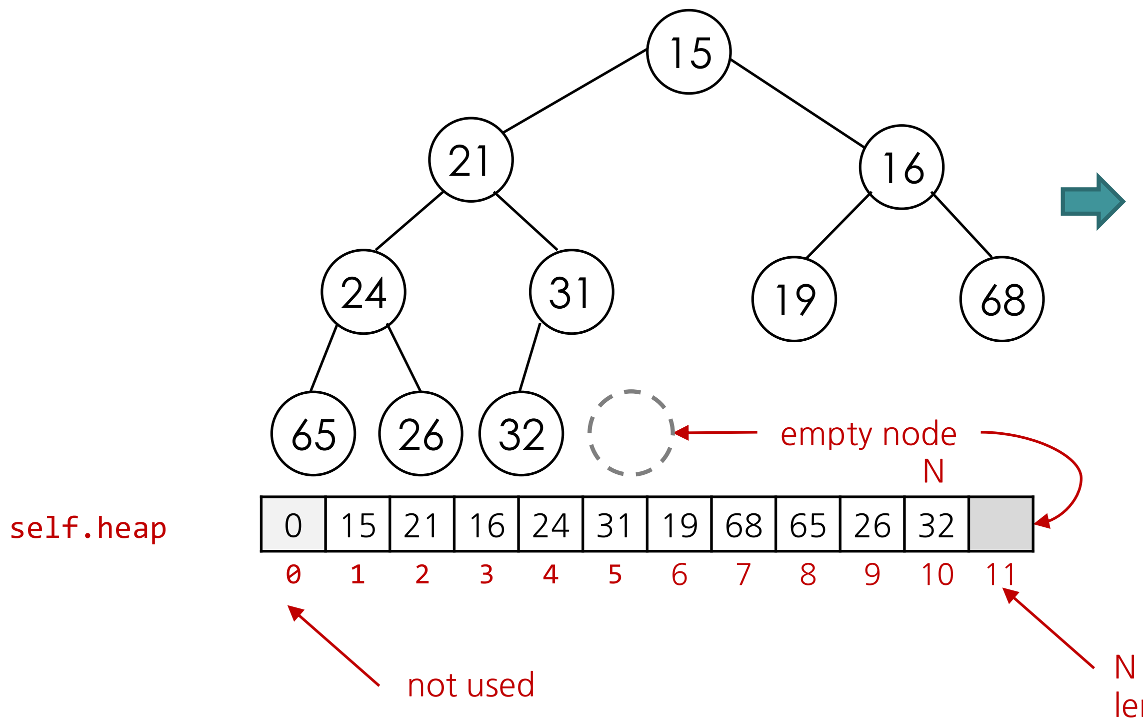
Agenda & Readings

- 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):                     # set more for min-heap, less for max-heap
        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
```



N = 10
len(self.heap) = 11
Notice the value N always
Use append() only if necessary

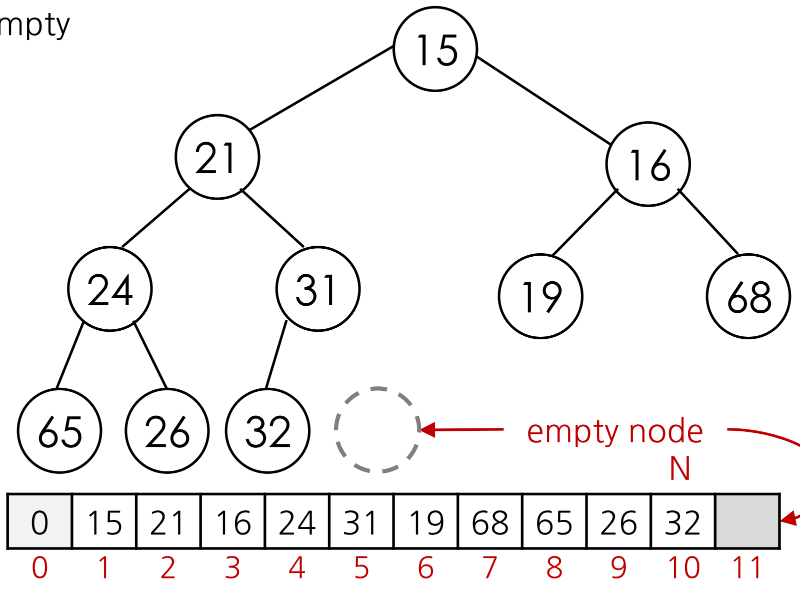
min-heap: insert(heap, 14)

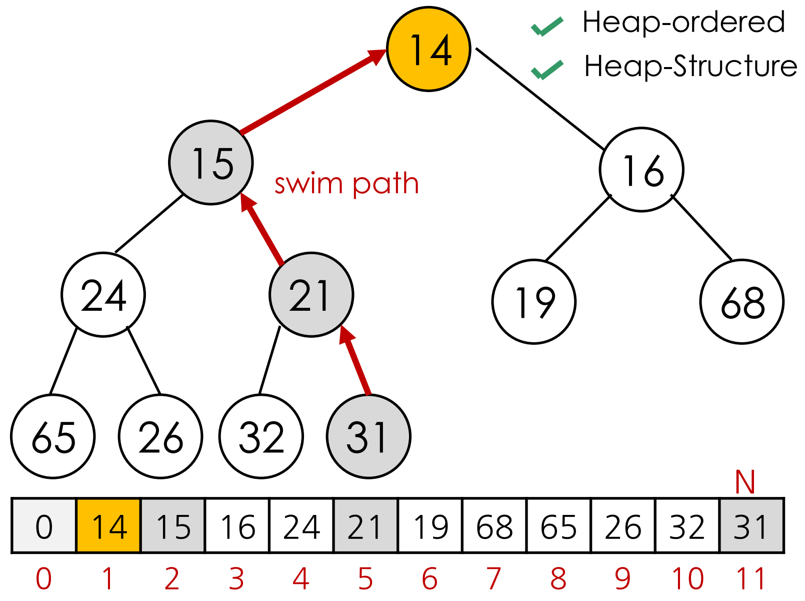
Algorithm:

- 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?



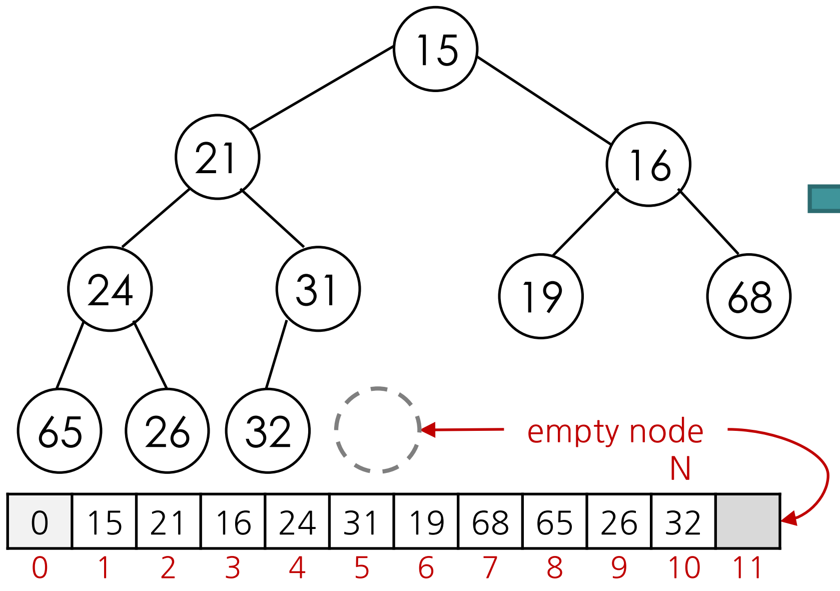


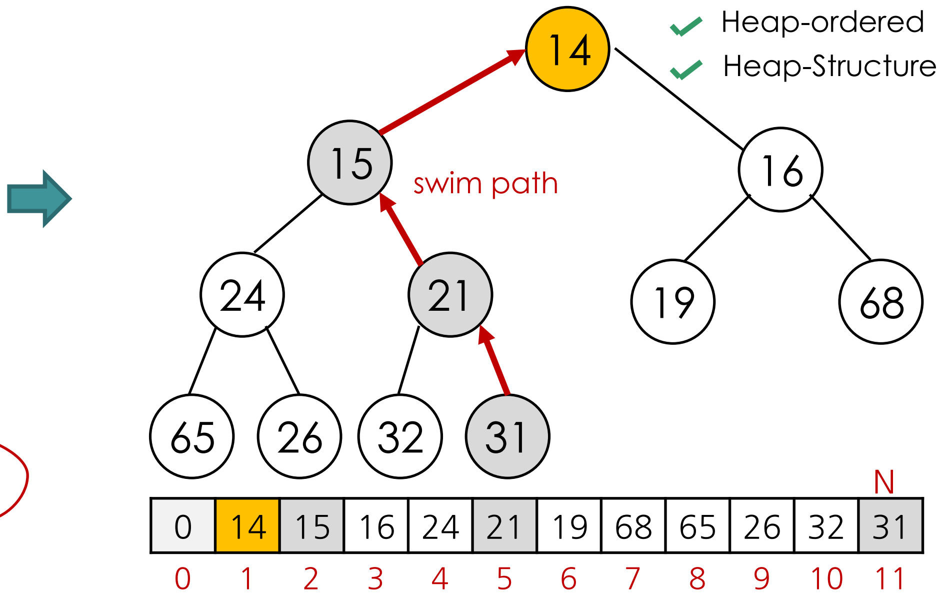
min-heap: insert(heap, 14)

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
```

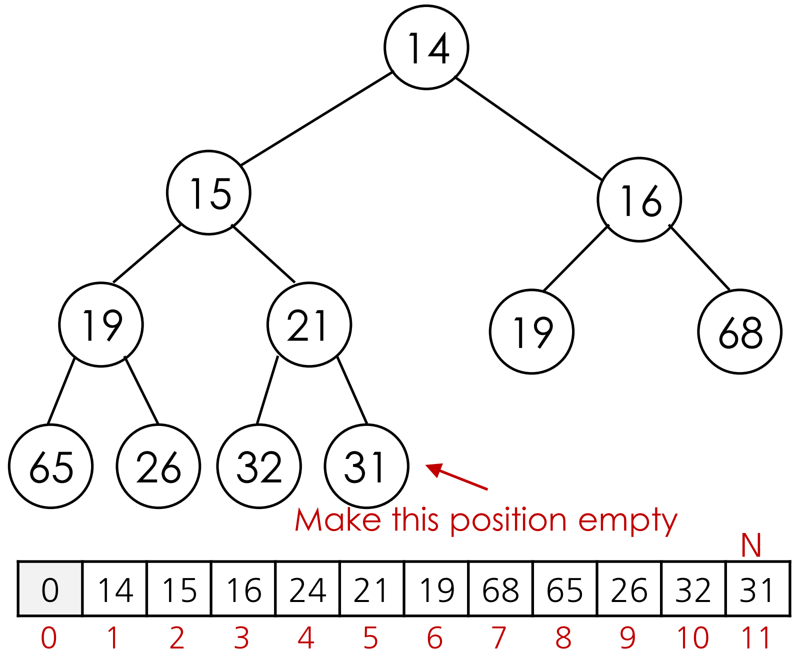




min-heap: delete() or dequeue()

Algorithm:

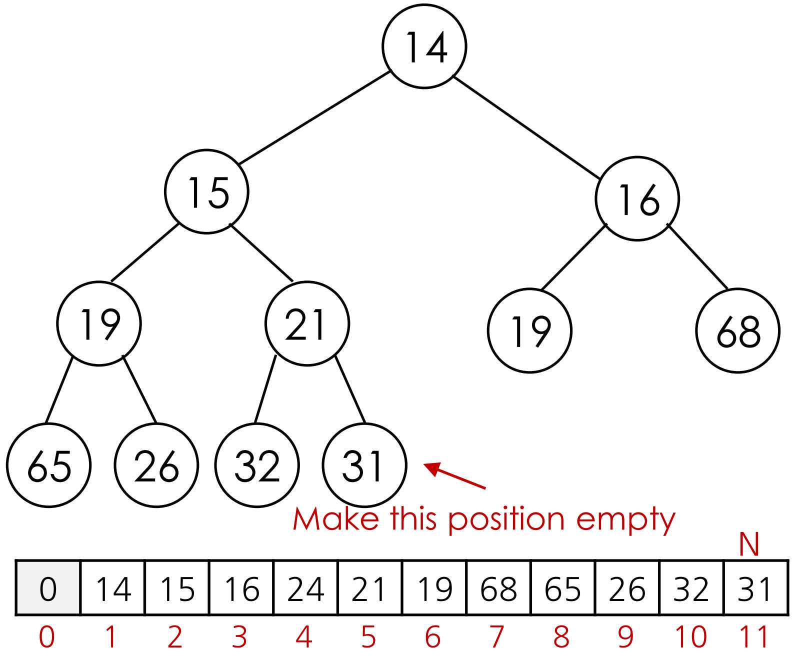
- Swap the root and the last element.
- Heap decreases by one in size.
- **Move down (sink) the root** while not satisfying heap-ordered.
    - Minimum element is always at the root (by min-heap definition).



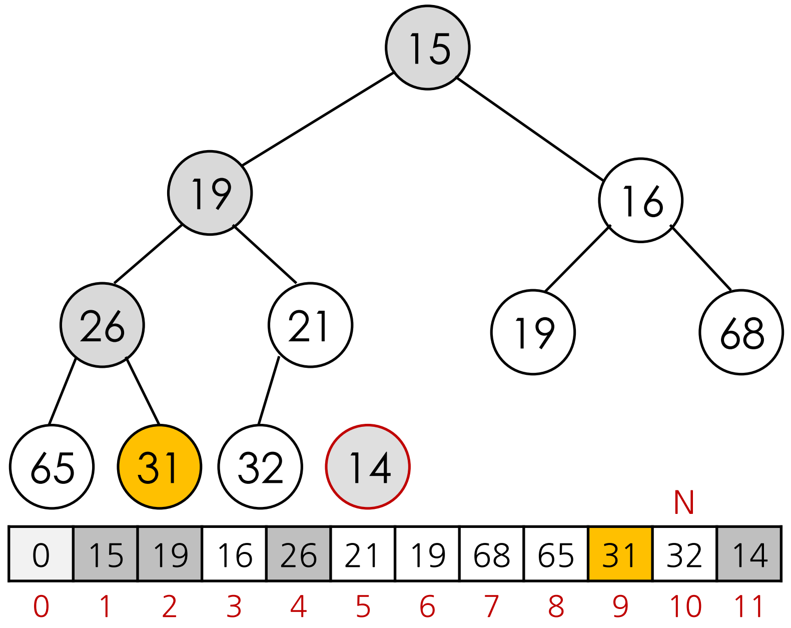
min-heap: delete() or dequeue()

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



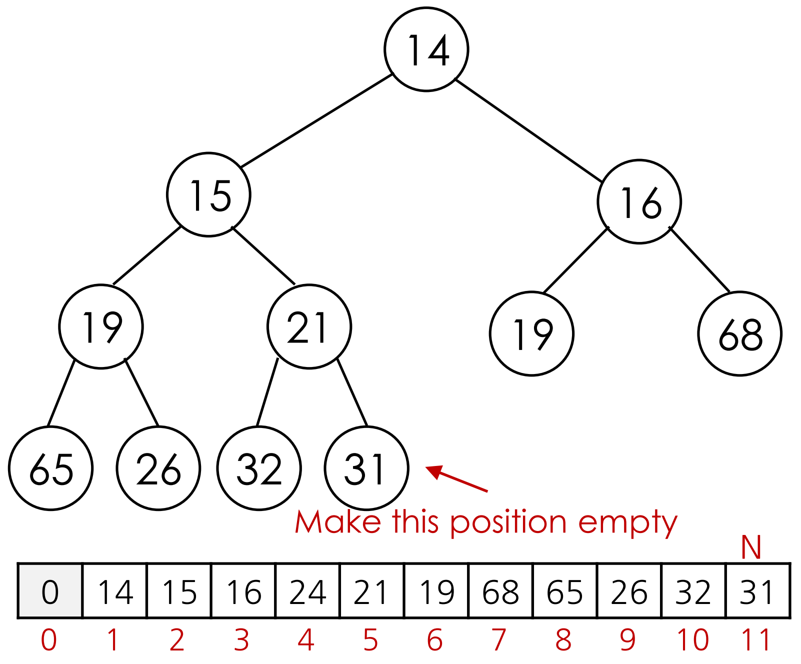
✓ Heap-ordered

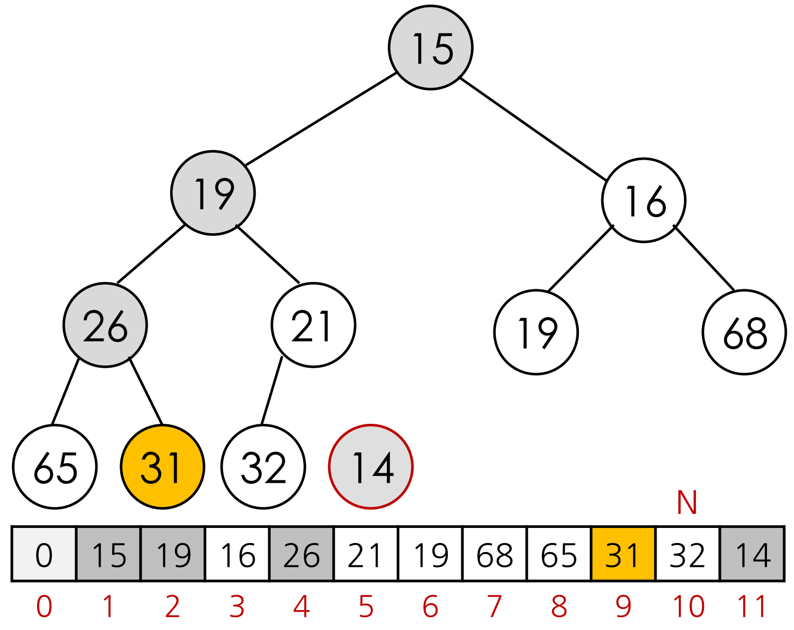
✓ Heap-Structure

min-heap: delete() or dequeue()

```
class BinHeap:
    ...
    def delete(self):
        retval = self.heap[1]                   # 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
```

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



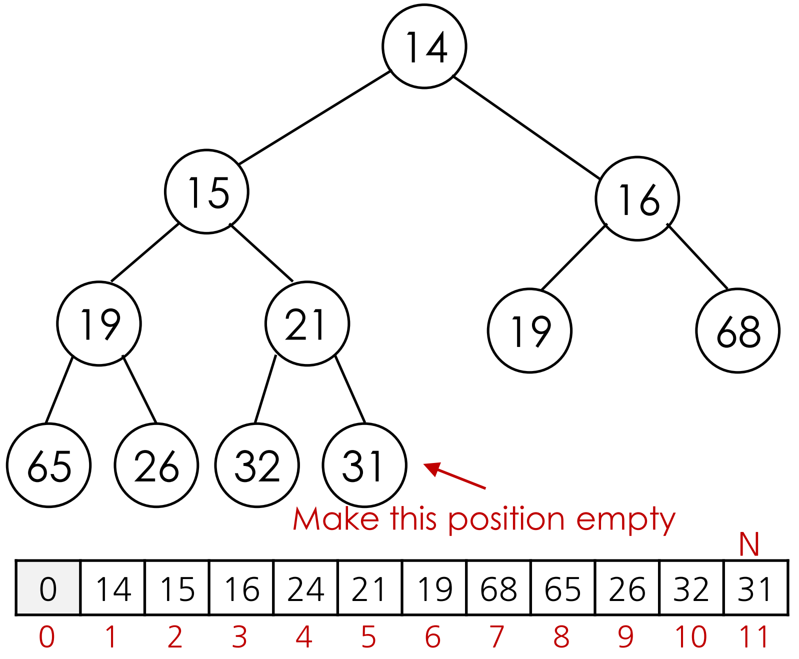


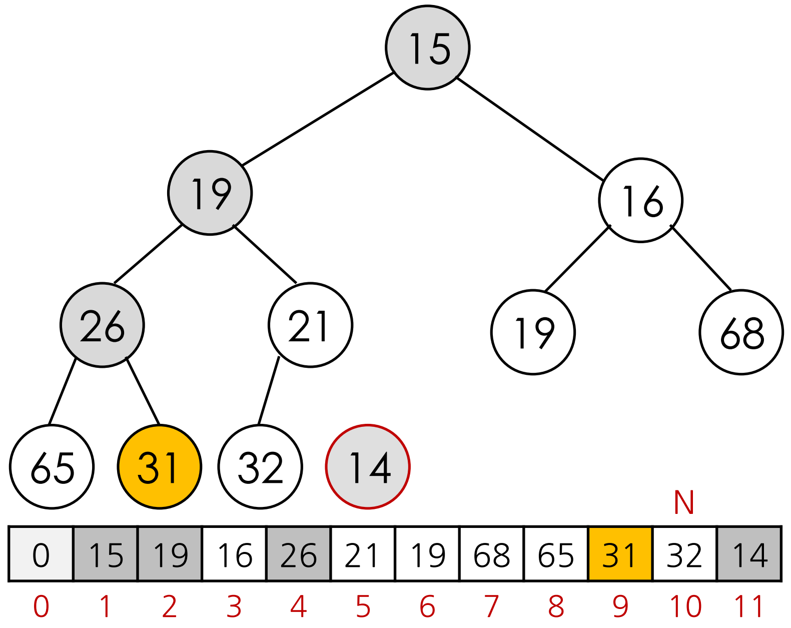
✓ Heap-ordered

✓ Heap-Structure

min-heap: delete() or dequeue()

```
class BinHeap:
    ...
    def sink(self, i):                                  # 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
```





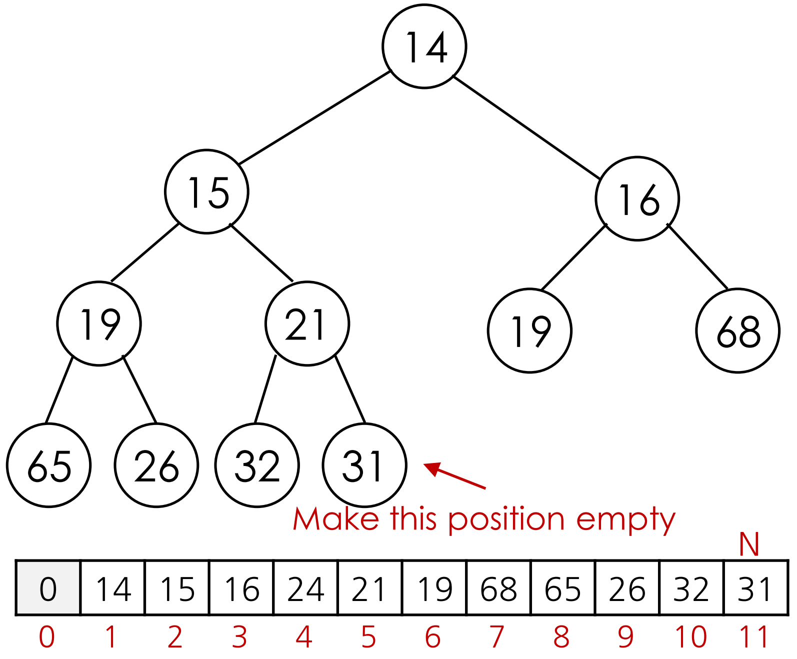
✓ Heap-ordered

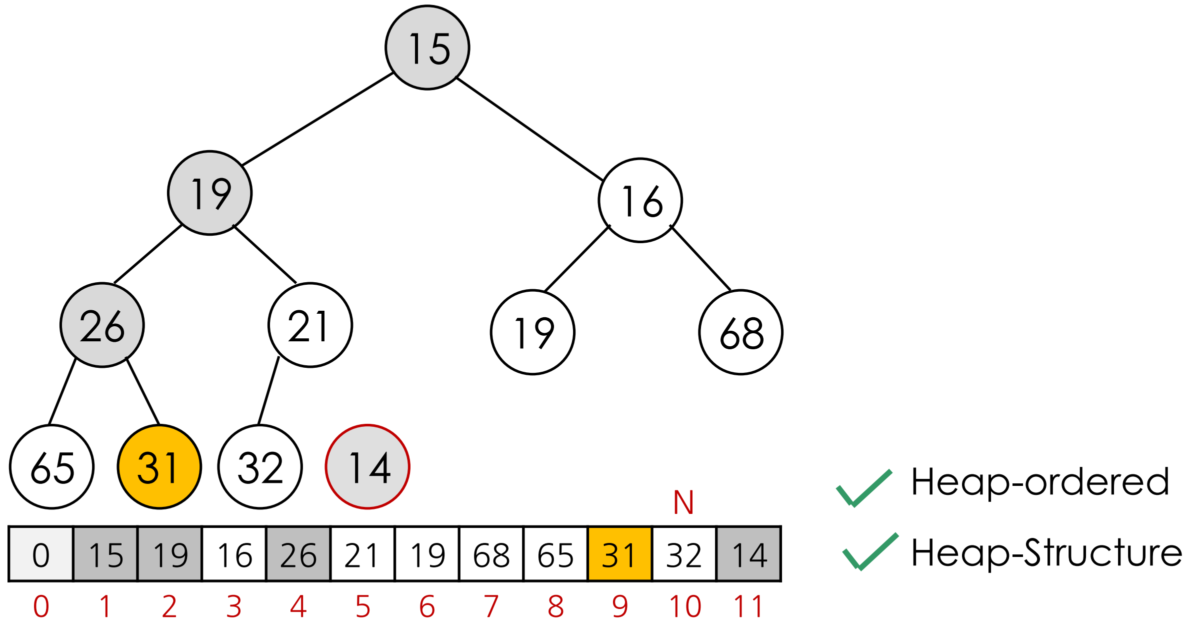
✓ Heap-Structure

min-heap: delete() or dequeue()

- What do you expect from the following code snippet?

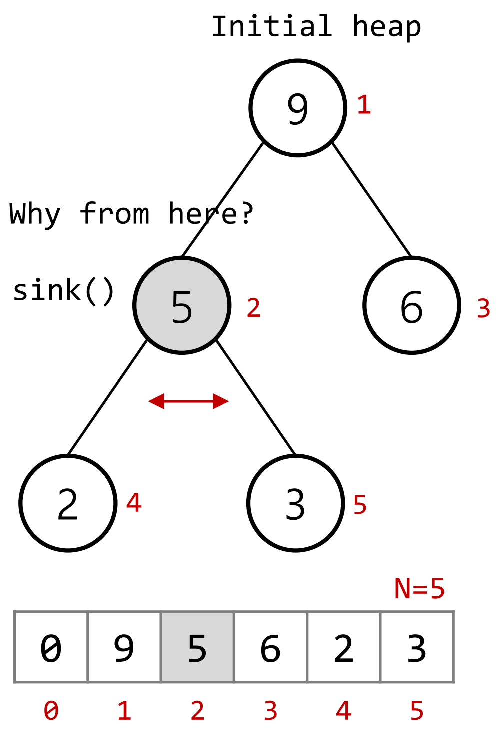
```
    result = [ bh.delete() for x in range(bh.N) ]
    print('              result:', result)
    print('number of elements N:', bh.N)
    print('  lengh of heap list:', len(bh.heap))
    print('    heap list stored:', bh.heap)
```





min-heap: buildHeap() and heapify()

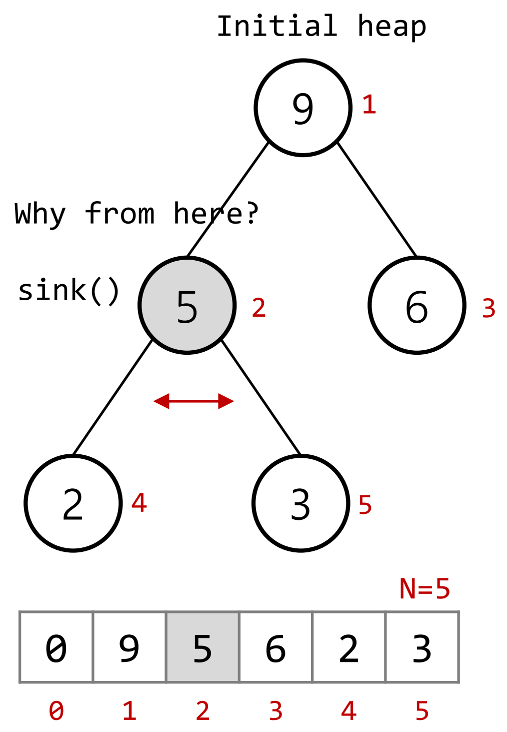
```
class BinHeap:
    ...
    def buildHeap(self, arr):          # build heap from input arr list
        self.heap = [0] + arr[:]       # set the initial heap
        self.N = len(arr)              # set the size
        i = len(arr) // 2              # get the last internal node
        while i > 0:                   # sink from the last internal node to root 1
            self.sink(i)
            i -= 1
```

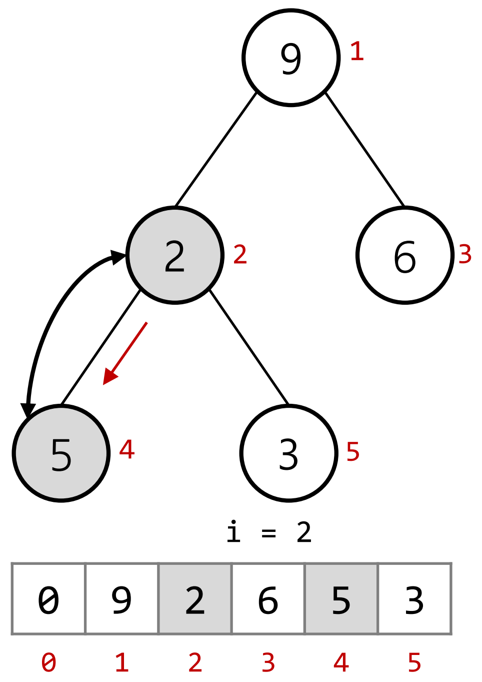


3

min-heap: buildHeap() and heapify()

```
class BinHeap:
    ...
    def buildHeap(self, arr):          # build heap from input arr list
        self.heap = [0] + arr[:]       # set the initial heap
        self.N = len(arr)              # set the size
        i = len(arr) // 2              # get the last internal node
        while i > 0:                   # sink from the last internal node to root 1
            self.sink(i)
            i -= 1
```

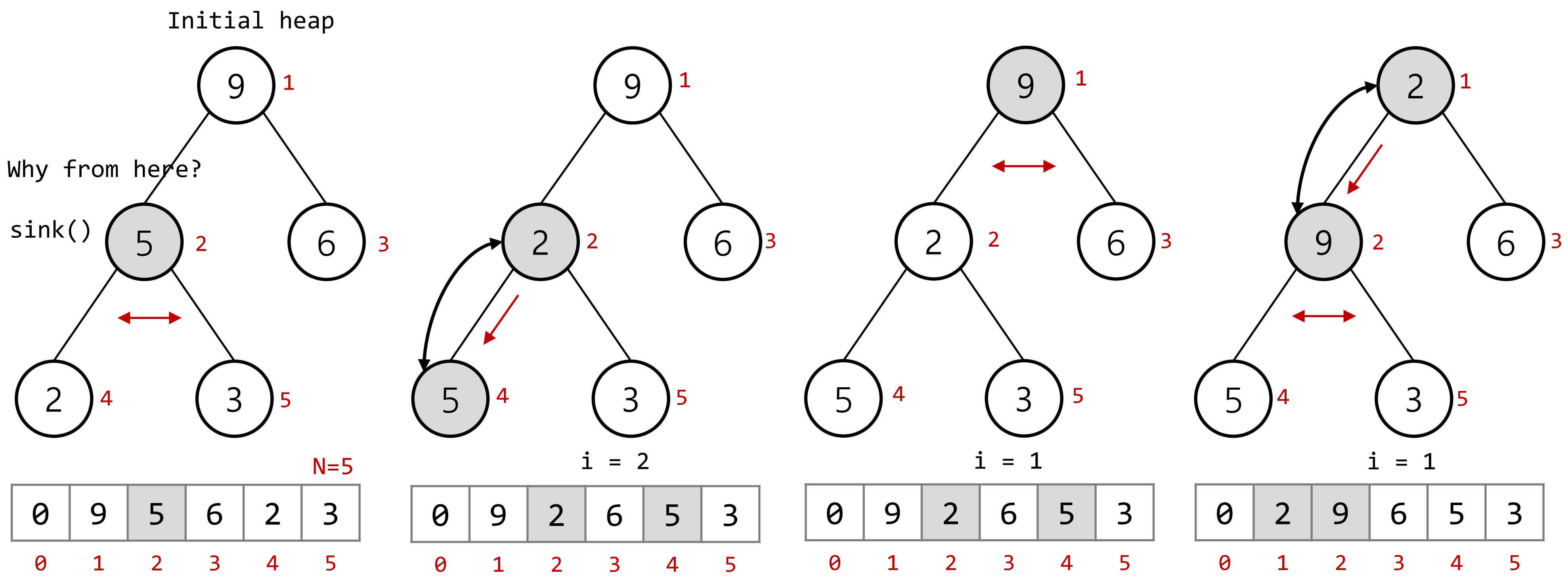




3

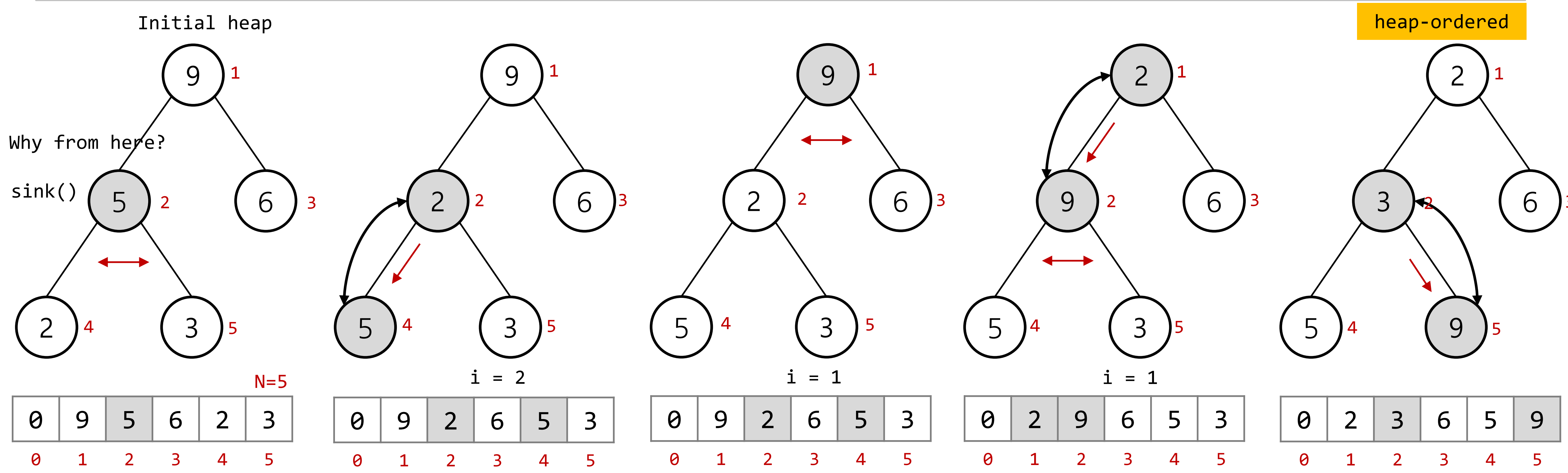
min-heap: buildHeap() and heapify()

```
class BinHeap:
    ...
    def buildHeap(self, arr):          # build heap from input arr list
        self.heap = [0] + arr[:]       # set the initial heap
        self.N = len(arr)              # set the size
        i = len(arr) // 2              # get the last internal node
        while i > 0:                   # sink from the last internal node to root 1
            self.sink(i)
            i -= 1
```



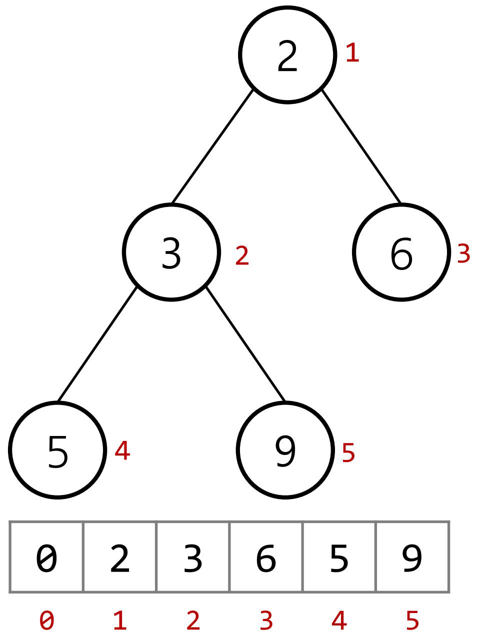
min-heap: buildHeap() and heapify()

```
class BinHeap:
    ...
    def buildHeap(self, arr):          # build heap from input arr list
        self.heap = [0] + arr[:]       # set the initial heap
        self.N = len(arr)              # set the size
        i = len(arr) // 2              # get the last internal node
        while i > 0:                   # sink from the last internal node to root 1
            self.sink(i)
            i -= 1
```



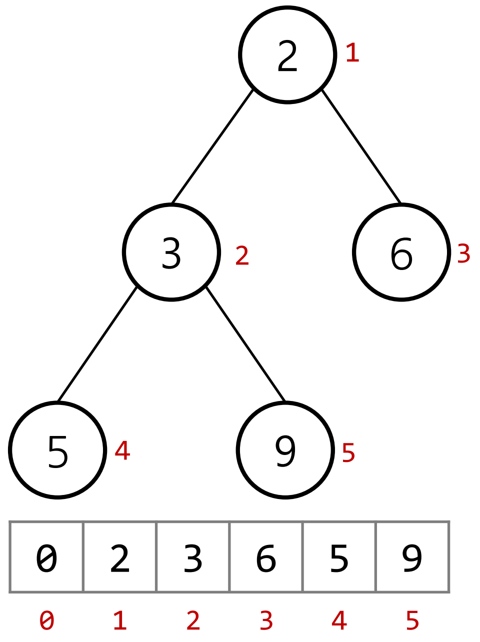
min-heap: buildHeap() and heapify()

```
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()

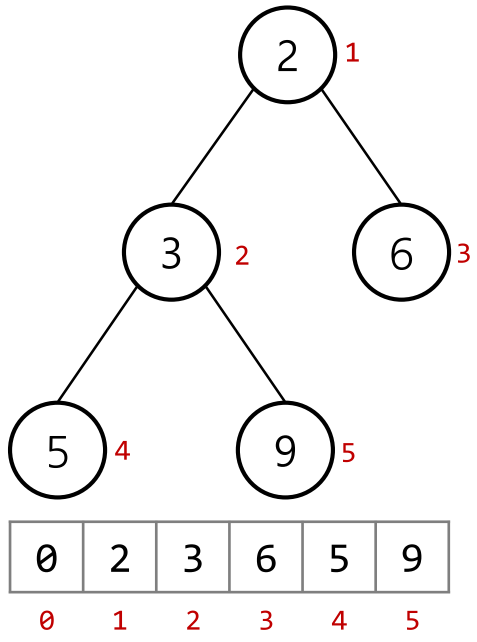
```
if __name__ == '__main__':
    bh = BinHeap()
    bh.buildHeap([9, 5, 6, 2, 3])
    print('number of elements N:', bh.N)
    print('  lengh of heap list:', len(bh.heap))
    print('    heap list stored:', bh.heap)
    bh.draw()

    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('  lengh 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('  lengh of heap list:', len(bh.heap))
    print('    heap list stored:', bh.heap)
```

```
number of elements N: 5
  lengh of heap list: 6
    heap list stored: [0, 2, 3, 6, 5, 9]
    2
   / \
  3   6
 / \
5   9

inserting: 7 - already heap-ordered
      2
     / \
    /   \
   /     \
  3       6
 / \     /
5   9   7

inserting: 1 - swim up, become root
      1
     / \
    /   \
   /     \
  3       2
 / \     / \
5   9   7   6

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

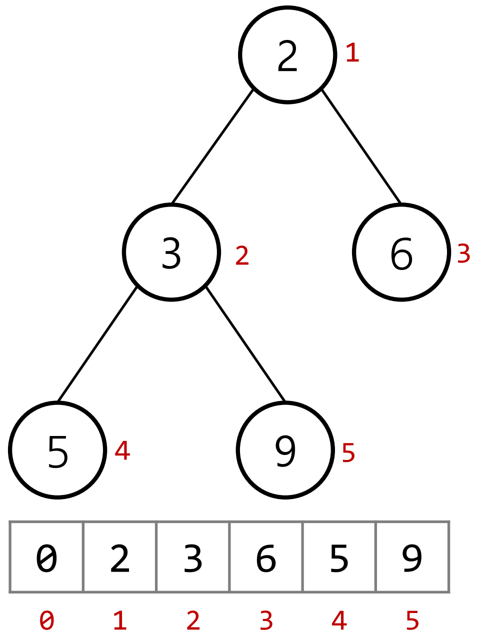


heap elements are deleted
need to change not to delete
to have .

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('  lengh 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('  lengh of heap list:', len(bh.heap))
    print('    heap list stored:', bh.heap)
```

```
number of elements N: 5
  lengh of heap list: 6
    heap list stored: [0, 2, 3, 6, 5, 9]
    2
   / \
  3   6
 / \
5   9

inserting: 7 - already heap-ordered
        2
       / \
      /   \
     /     \
    3       6
   / \     /
  5   9   7

inserting: 1 - swim up, become root
        1
       / \
      /   \
     /     \
    3       2
   / \     / \
  5   9   7   6
```



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

heap sort

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

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

Binary heap: Time complexity:

- Level of heap is $\lfloor \log_2 N \rfloor$
- insert: O(log N) for each insert
  - In practice, expect less
- delete: O(log N) // deleting root node or any node
- increase/decrease key: O(log N)

Implementation	Insert	Delete	max
Unordered array	1	N	N
Ordered array	N	1	1
Binary heap	**log N**	**log N**	1

$\uparrow$ $\uparrow$

Mission Completed

Data Structures in Python

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