HEAP

Tian Zhang
2018/09/09

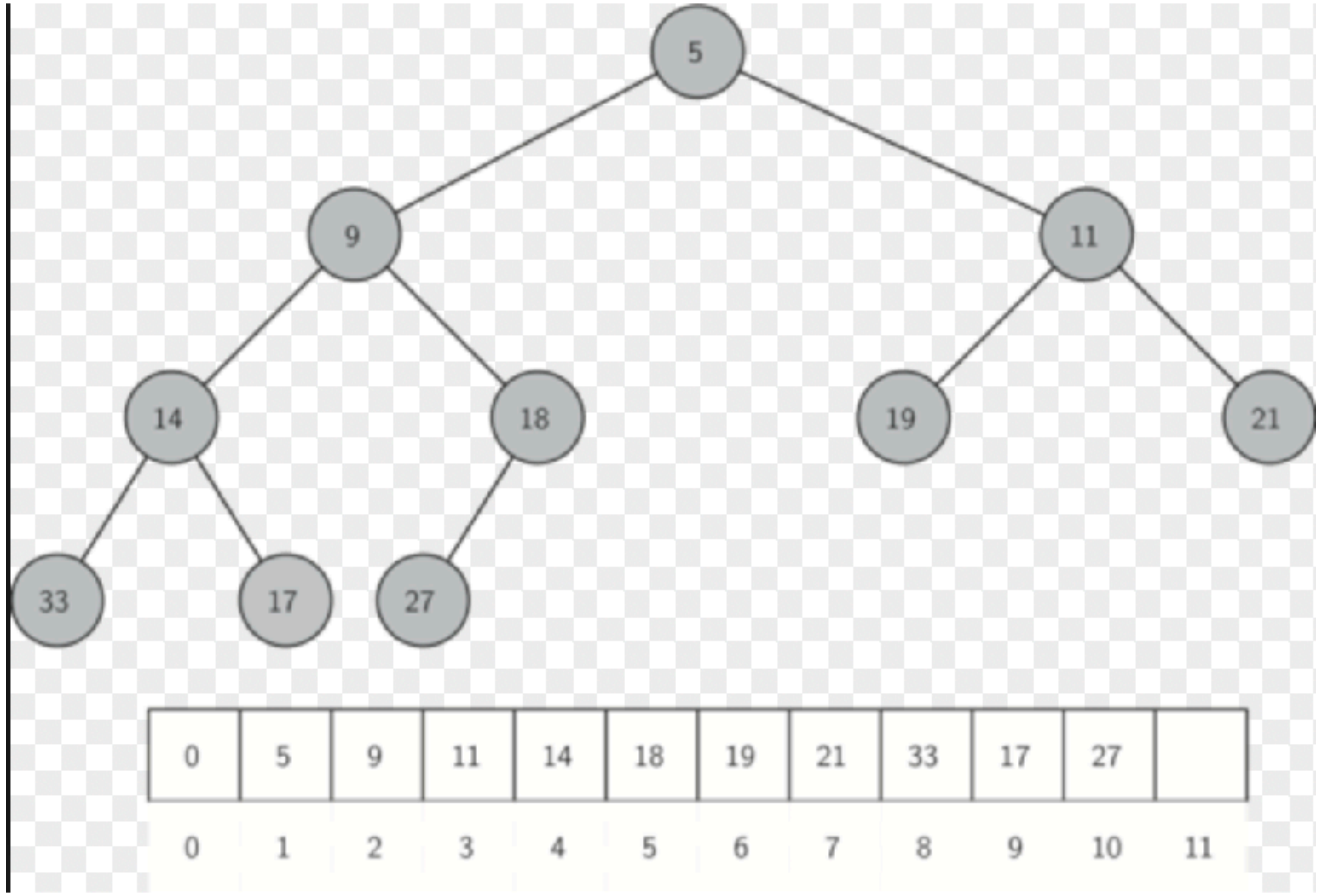
OVERVIEW

- Introduction
- Implementation
- Application

INTRODUCTION

- Heap is a specialized tree-based data structure that satisfies the heap property
- The **min**-heap property: the value of each node is greater than or equal to the value of its parent, with the minimum-value element at the root.
- The **max**-heap property: the value of each node is less than or equal to the value of its parent, with the maximum-value element at the root.
- Heap sometimes refer as priority queue, since PQ is very easy to implement from heap and they have similar characteristics, i.e. when you enqueue an item on a priority queue, the new item may move all the way to the front.

Complete Binary Tree



IMPLEMENTATION

- A common implementation is the binary heap
- ***Array*** based heap outperforms ***Linked-list*** based heap
    I. Easy to compute -> if parent location is p, then children location is 2p and 2p+1
    II. Lower memory usage -> no pointer needed
    III. Easier memory management -> only one object allocated rather than N

IMPLEMENTATION

- **BinaryHeap():** creates a new, empty, binary heap.
- **size():** returns the number of items in the heap.
- **insert(k):** adds a new item to the heap.
- **findMin():** returns the item with the minimum key value, leaving item in the heap.
- **delMin():** returns the item with the minimum key value, removing the item from the heap.
- **buildHeap(list):** builds a new heap from a list of keys.

IMPLEMENTATION - insert(k)

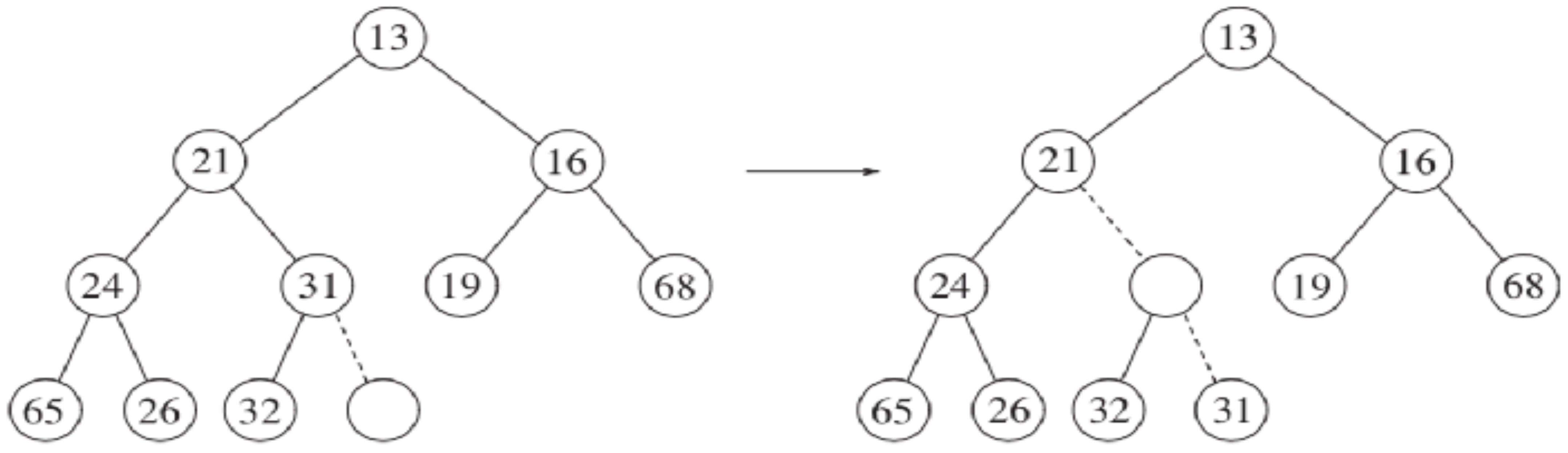


Figure 6.6 Attempt to insert 14: creating the hole, and bubbling the hole up

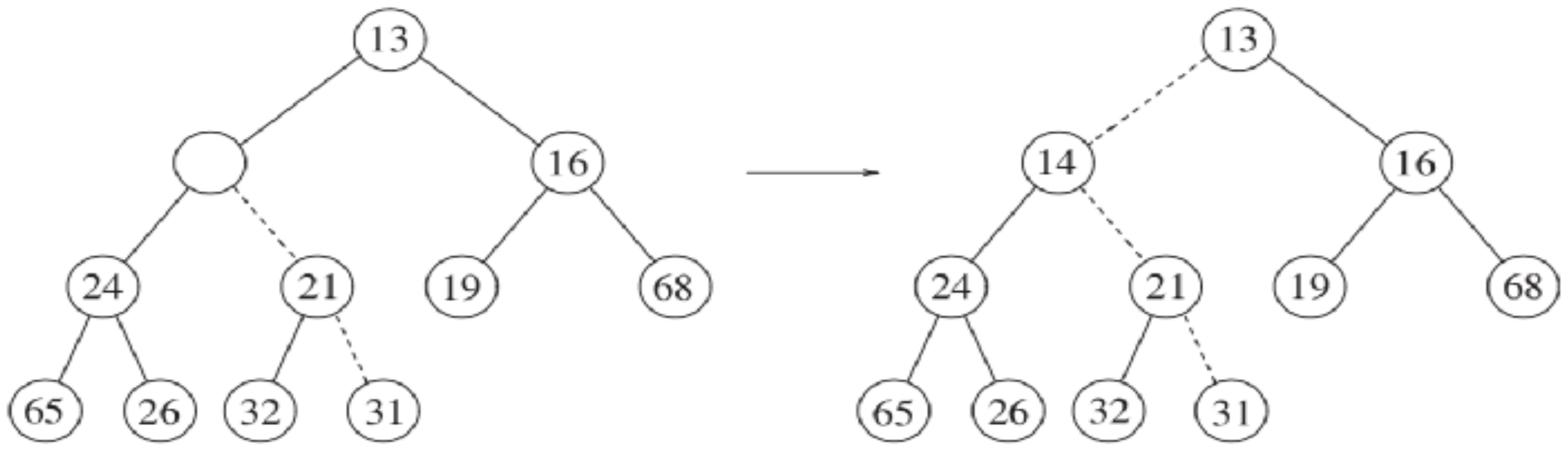
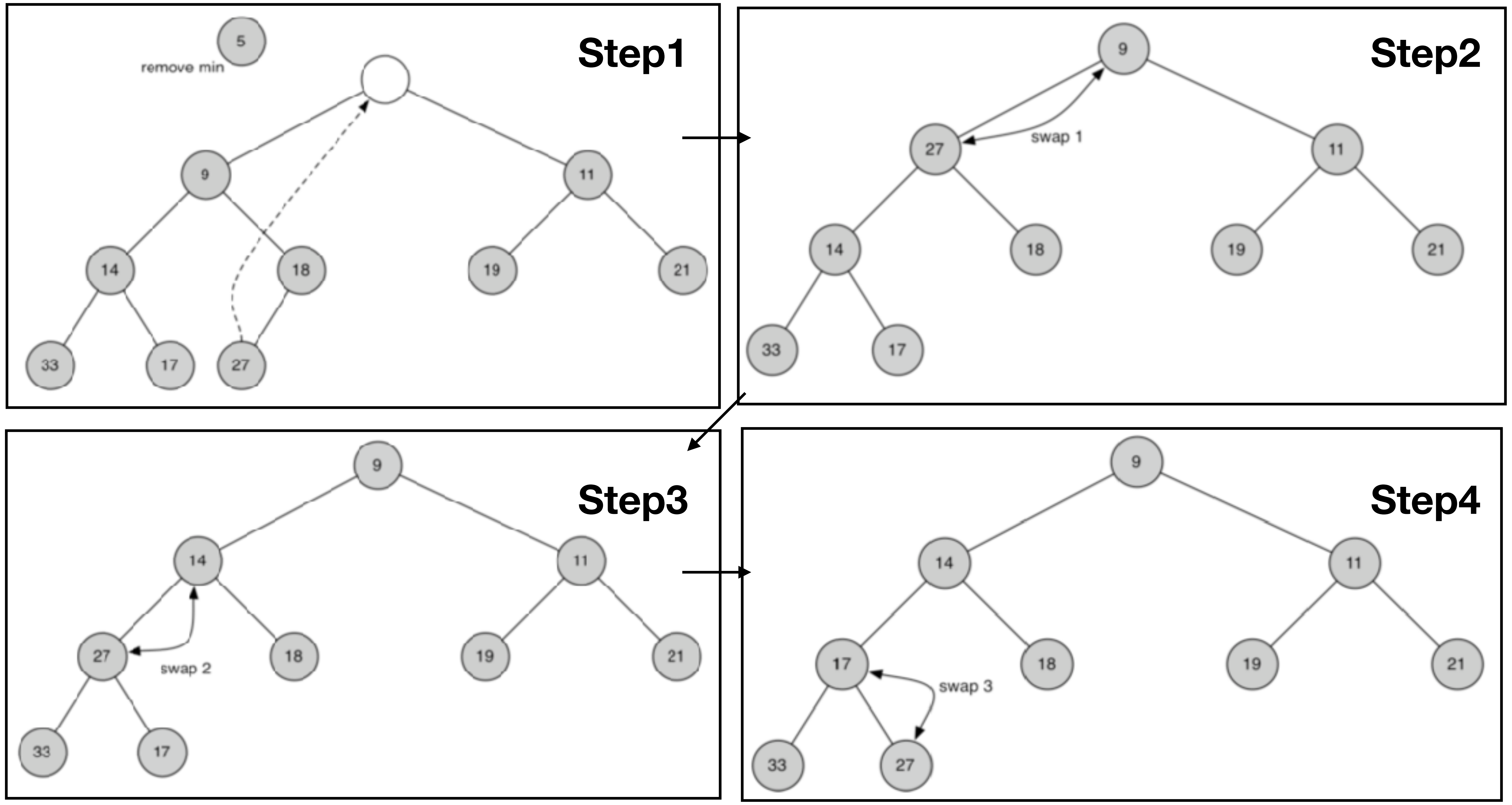


Figure 6.7 The remaining two steps to insert 14 in previous heap

Complexity: the maximum height of tree is log(n), so when Insert an element, we need to percolate up by doing maximum log(n) swap operations, thus O(log(n))

```python
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0

    def insert(self,k):
        self.heapList.append(k)
        self.currentSize = self.currentSize + 1
        self.perUp(self.currentSize)

    def perUp(self,p):
        while p // 2 > 0:
            if self.heapList[p] < self.heapList[p // 2]:
                self.heapList[p],self.heapList[p // 2] =
                    self.heapList[p // 2], self.heapList[p]
                p = p // 2
            else:
                break

    def findMin(self):
        if self.currentSize == 0:
            return None
        else:
            return self.heapList[1]
```

IMPLEMENTATION - delMin()



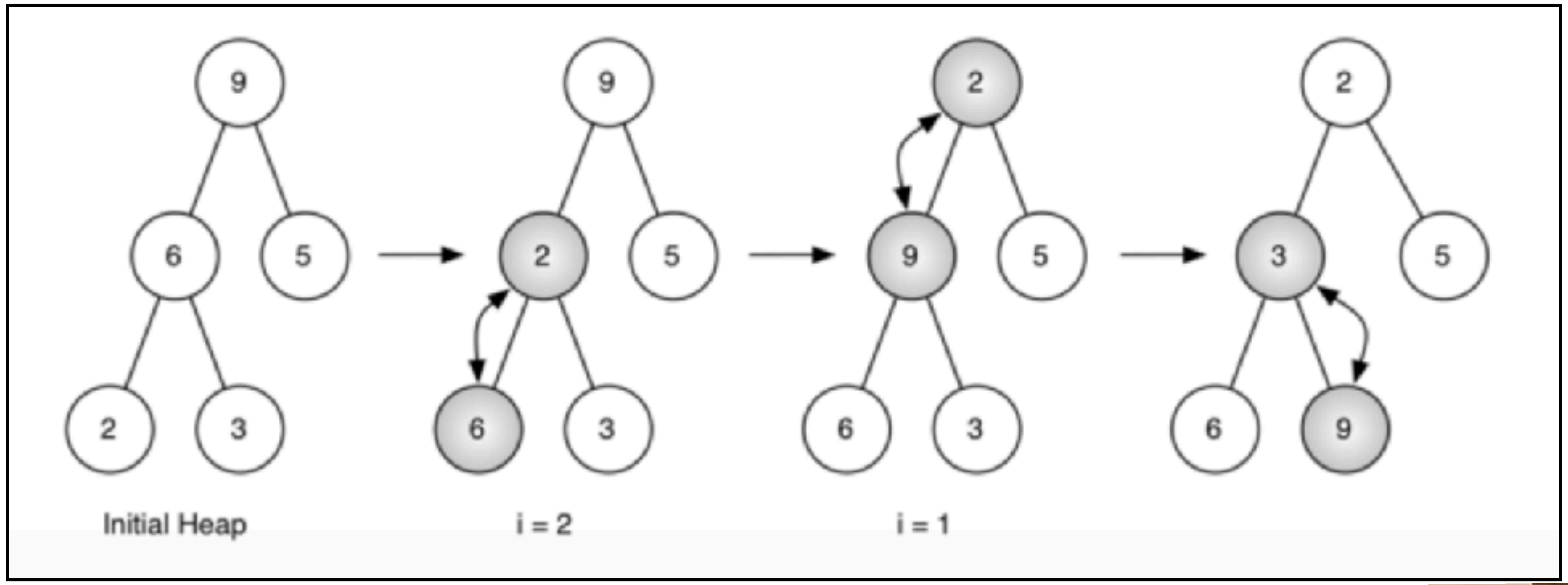
Complexity: the maximum height of tree is log(n), so after extracting the minimum element, we need to percolate down by doing maximum log(n) swap operations to ensure the heap property still works, thus O(log(n))

```
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0

    def delMin(self):
        if self.currentSize == 0:
            return None
        retval = self.heapList[1]
        self.heapList[1] = self.heapList[self.currentSize]
        self.currentSize = self.currentSize - 1
        self.heapList.pop()
        self.perDown(1)
        return retval
```

Heapify

```
    def perDown(self,p):
        while p*2 <= self.currentSize:
            mc = self.minChild(p)
            if self.heapList[p] > self.heapList[mc]:
                self.heapList[p], self.heapList[mc] =
                        self.heapList[mc], self.heapList[p]
            p = mc

    def minChild(self,p):
        if p * 2 + 1 > self.currentSize:
            return p * 2
        else:
            if self.heapList[p*2] < self.heapList[p*2+1]:
                return p*2
            else:
                return p*2+1
```

IMPLEMENTATION - buildHeap(list)



Complexity: O(n)

1 2 4 $\boxed{8}$ $\boxed{16}$ leaves

.... $\frac{N}{8}$ $\frac{N}{4}$ $\frac{N}{2}$

$$\frac{N}{4}\cdot(1c) + \frac{N}{8}(2c) + \frac{N}{16}(3c) + \cdots 1\cdot(\log N\, c).$$

Set $\frac{N}{4} = 2^k$

$$c\cdot 2^k\left(\frac{1}{2^0} + \frac{2}{2^1} + \frac{3}{2^2} + \cdots + \frac{k+1}{2^k}\right)$$

$$= c\cdot\frac{N}{4}\underbrace{\boxed{\sum_{i=0}^{k}\frac{i+1}{2^i}}}_{\text{Constant}} \propto N$$

Heapify

```python
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0

    def perDown(self,p):
        while p*2 <= self.currentSize:
            mc = self.minChild(p)
            if self.heapList[p] > self.heapList[mc]:
                self.heapList[p], self.heapList[mc]
                = self.heapList[mc], self.heapList[p]
            p = mc

    def minChild(self,p):
        if p * 2 + 1 > self.currentSize:
            return p * 2
        else:
            if self.heapList[p*2] < self.heapList[p*2+1]:
                return p*2
            else:
                return p*2+1

    def buildHeap(self,alist):
        p = len(alist)//2
        self.currentSize = len(alist)
        self.heapList = [0] + alist[:]
        while (p>0):
            self.perDown(p)
            p -= 1
```

APPLICATION

- HeapSort
- Find the kth smallest element
- Merge k Sorted Lists

APPLICATION: HeapSort

Pseudo algorithm

1. Build min_heap from unordered array - O(n)

2. Find min element and add to sorted array - O(1)

3. Discard minimum element from the heap - O(lgn)

4. Repeat 2-3 until heap size is 0 - O(n)

Complexity

O(nlogn)

Question

How to do it in-place?

```python
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0

    def perDown(self,p):
        while p*2 <= self.currentSize:
            mc = self.minChild(p)
            if self.heapList[p] > self.heapList[mc]:
                self.heapList[p], self.heapList[mc] = self.heapList[mc], self.heapList[p]
            p = mc

    def minChild(self,p):
        if p * 2 + 1 > self.currentSize:
            return p * 2
        else:
            if self.heapList[p*2] < self.heapList[p*2+1]:
                return p*2
            else:
                return p*2+1

    def buildHeap(self,alist):
        p = len(alist)//2
        self.currentSize = len(alist)
        self.heapList = [0] + alist[:]
        while (p>0):
            self.perDown(p)
            p -= 1
    def delMin(self):
        if self.currentSize == 0:
            return None
        retval = self.heapList[1]
        self.heapList[1] = self.heapList[self.currentSize]
        self.currentSize = self.currentSize - 1
        self.heapList.pop()
        self.perDown(1)
        return retval

    def heapSort(self,alist):
        aheap = self.buildHeap(alist)
        sorted_alist = []
        while self.currentSize > 0:
            sorted_alist.append(self.delMin())
        return sorted_alist
```

APPLICATION: Find kth smallest element

Pseudo algorithm

1. Build min_heap from unordered array - O(n)

2. Extract min element k times from min heap - O(klogn)

Complexity

O(n+klogn)

```python
class BinHeap:
    def __init__(self):
        self.heapList = [0]
        self.currentSize = 0

    def perDown(self,p):
        while p*2 <= self.currentSize:
            mc = self.minChild(p)
            if self.heapList[p] > self.heapList[mc]:
                self.heapList[p], self.heapList[mc] = self.heapList[mc], self.heapList[p]
            p = mc

    def minChild(self,p):
        if p * 2 + 1 > self.currentSize:
            return p * 2
        else:
            if self.heapList[p*2] < self.heapList[p*2+1]:
                return p*2
            else:
                return p*2+1

    def buildHeap(self,alist):
        p = len(alist)//2
        self.currentSize = len(alist)
        self.heapList = [0] + alist[:]
        while (p>0):
            self.perDown(p)
            p -= 1
    def delMin(self):
        if self.currentSize == 0:
            return None
        retval = self.heapList[1]
        self.heapList[1] = self.heapList[self.currentSize]
        self.currentSize = self.currentSize - 1
        self.heapList.pop()
        self.perDown(1)
        return retval

    def findKsmall(self,alist,k):
        aheap = self.buildHeap(alist)
        N = self.currentSize
        while self.currentSize != N-k:
            res = self.delMin()
        return res
```

APPLICATION: Merge k Sorted Lists (LC 23)

Merge *k* sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

Example:

```
Input:
[
  1->4->5,
  1->3->4,
  2->6
]
Output: 1->1->2->3->4->4->5->6
```

```python
class Solution:
    def mergeKLists(self, lists):
        """
        :type lists: List[ListNode]
        :rtype: ListNode
        """
        import heapq
        heapList = []
        for lst in lists:
            while lst:
                heapq.heappush(heapList, lst.val)
                lst = lst.next

        head = ListNode(0)
        cur = head

        while heapList:
            cur.next = ListNode(heapq.heappop(heapList))
            cur = cur.next

        return head.next
```

Assignment

1. Implement Max-heap

2. Use max-heap to implement **in-place** HeapSort

3. LC 295: https://leetcode.com/problems/find-median-from-data-stream/description/

4. LC 264: https://leetcode.com/problems/ugly-number-ii/description/

5. LC 239: https://leetcode.com/problems/sliding-window-maximum/description/

6. LC 313: https://leetcode.com/problems/super-ugly-number/description/