```
Algorithm Loop2(n)
        s ← 0
        for i ← 1 to n^3 do
            for j ← 1 to i do
                s ← s + i
```

When we analyze the Big O for this code, it is going to be O(n^6).

The outer loop executes n^3,

Inner loop executes on average n^3 / 2 = O(n^3)

Total Complexity is n^3 * n ^3 = n ^ 6.

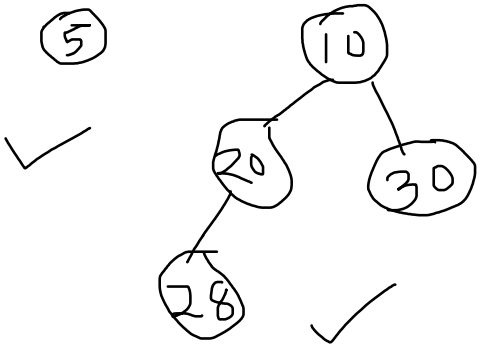
Lesson-6-Heap

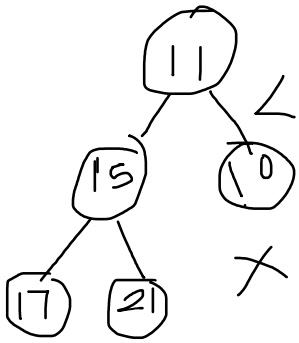
Heap is a hierarchical data structure and it's binary tree. It may be a min-heap or max-heap.

Min-heap - ***key***(***v***) >= ***key***(***parent***(***v***))

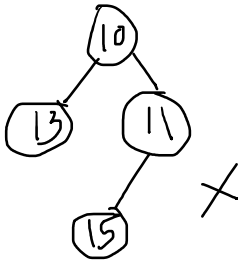
That is, the key of every child node is greater than or equal to the key(data you insert is the key) of it's parent node. Heap is a Binary tree.

Say given trees are valid min-heap or not.





child 10 < 11 root.



You have insert from the left at last level.

Rules

1. Heap is a Binary tree.
2. If heap is empty add as root.
3. The next nodes are inserted from left to right.
4. After inserting each node check the heap property. If any violation, restore the property by doing upheap.

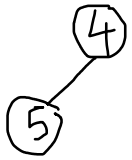
Construct a Min-heap with the following data,

4 5 3 7 6 9 1

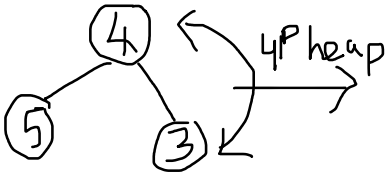
Min-Heap Insertion Step by Step (inserting from root to external node(leaf) – Top down approach

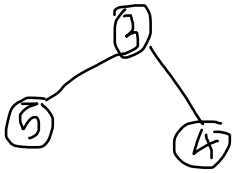
Insert 4, tree is empty. 4 becomes root.

Insert. 5 as the left child of 4

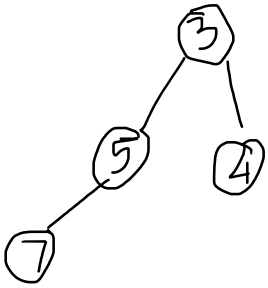


Insert 3, it violate the heap property, you have restore the property by doing upheap.

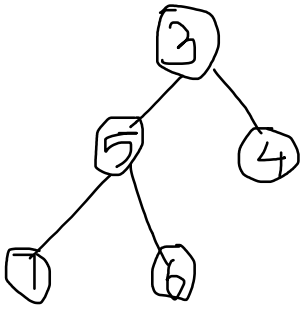




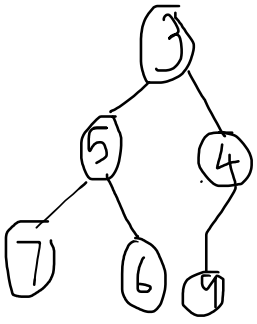
Insert 7



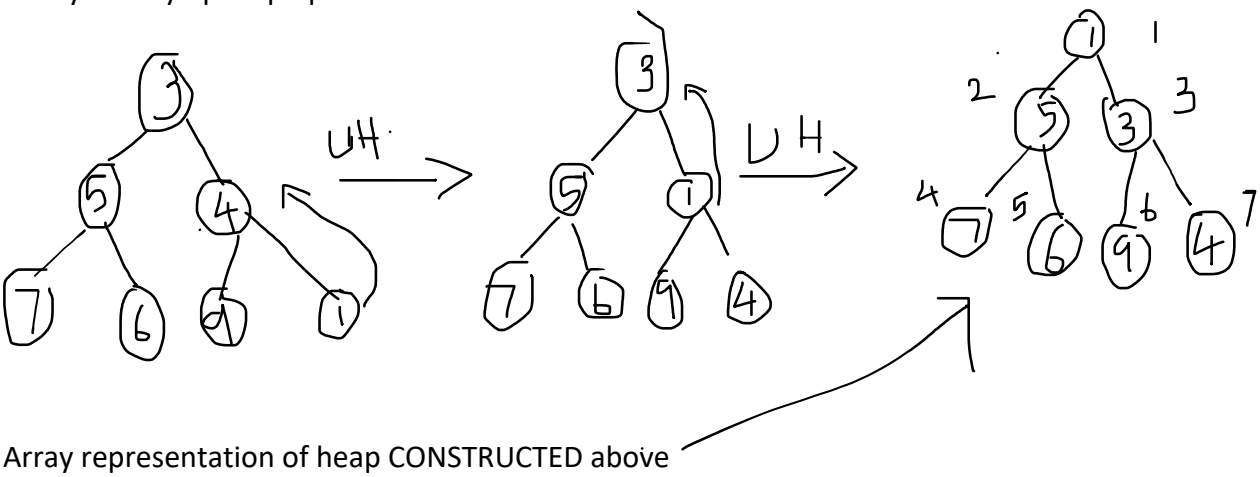
Insert 6



Insert 9



Insert 1 (Violates the heap property, you have to upheap until the right position. In the worst case you may upheap up to the root.



Array representation of heap CONSTRUCTED above

- Array length is n+1. n is total number of nodes.
- Index 0 is empty and not used.
- Root node is at index 1.
- Left child is at 2*its parent index.
- Right child is at 2*its parent index+1.

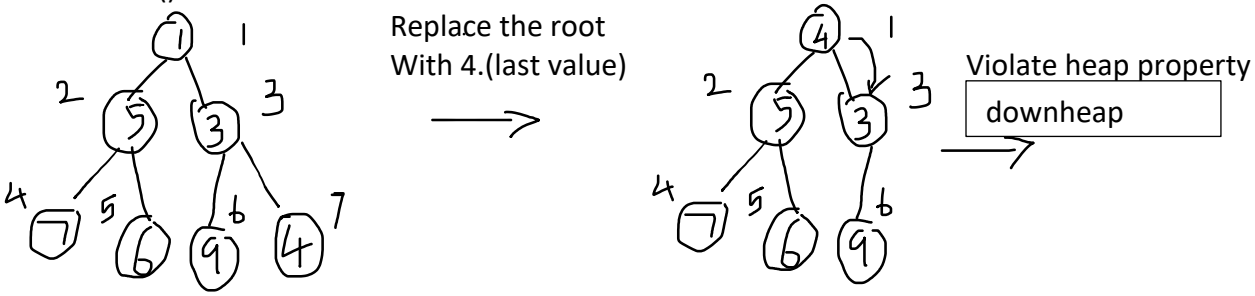
EMPTY	1	5	3	7	6	9	4
Index 0	1	2	3	4	5	6	7

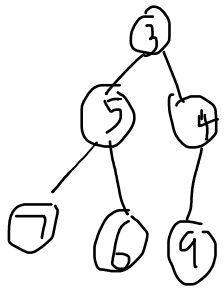
Min-heap Deletion

Rules

1. Remove the min from the root. You can't delete any other element randomly.
2. Replace the root with the last element in the array.
3. After replaced the root, check the heap property. If any violation, restore the property by doing down heap.

removeMin()





Analysis

Upheap():

- This operation is used when a new element is inserted into the heap, and it may violate the heap property by being smaller (in a min-heap) than its parent.
- The element is compared with its parent and possibly swapped, and this process is continued up the heap until the heap property is restored.
- Runtime Complexity: O(log n), since in the worst case, you might need to compare and swap all the way from the leaf level to the root.

Downheap():

- This operation is used when the root element is removed (in the case of removeMin()) and the last element in the heap is temporarily moved to the root. The heap property may be violated, so this element is compared with its children and swapped with the smaller child until the property is restored.
- Runtime Complexity: O(log n), because you may need to compare and swap the element at each level of the heap's height.

insertElem(element):

- To insert an element, it is added to the end of the array (which represents the heap), and then Upheap() is called to restore the heap property.

- Runtime Complexity: O(logn), which comes from the Upheap() operation as the actual insertion is O(1), but the restoring of the heap property is O(log n).

removeMin() (in a min-heap):

- This involves removing the root element, which is the minimum in the heap. The last element in the heap is moved to the root, and then Downheap() is called to restore the heap property.

- Runtime Complexity: O(log n), since the removal of the root element is O(1), but the Downheap() operation is O(log n).

In each of these operations, most of the work involves restoring the heap property by moving an element up or down the tree, and because the height of the tree is O(log n), that dominates the runtime complexity.

Min-heap Inserting element

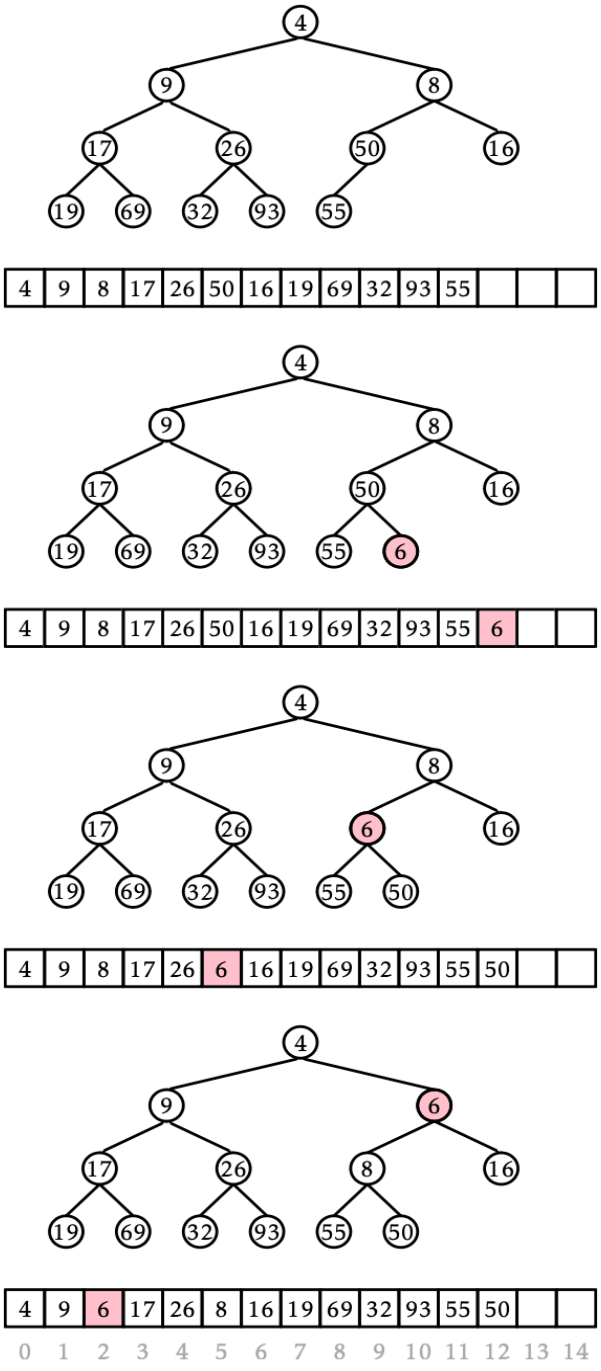


Figure 10.2: Adding the value 6 to a BinaryHeap.

Min-Heap Removal

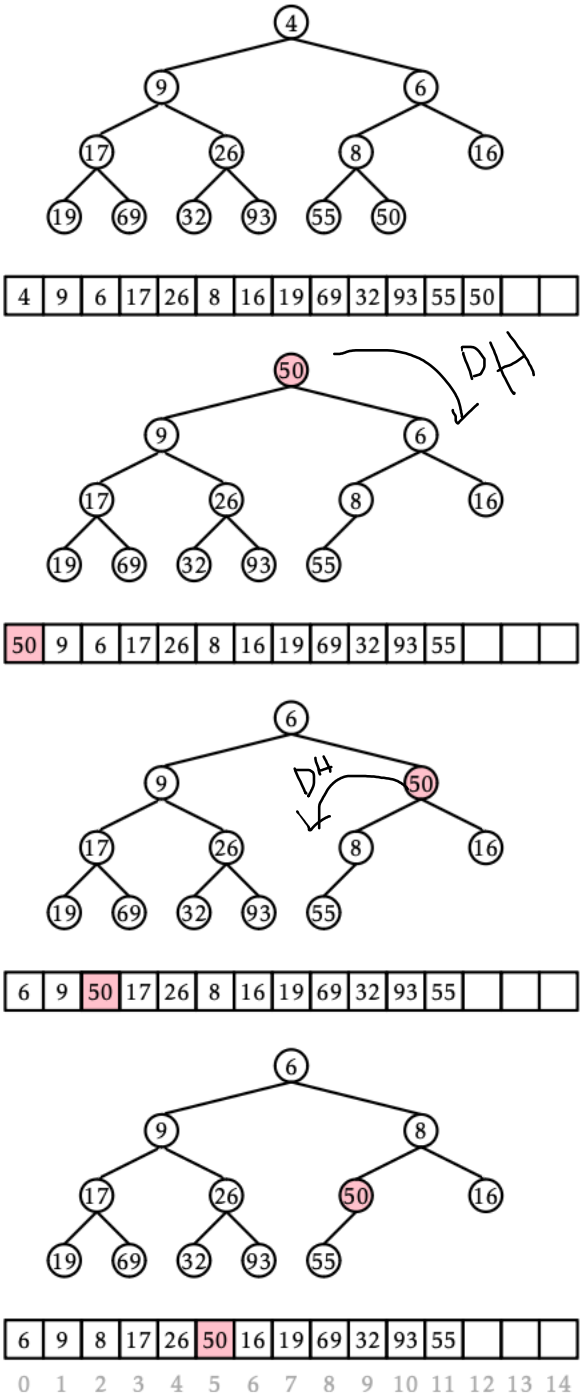


Figure 10.3: Removing the minimum value, 4, from a `BinaryHeap`.

Slide – 22 – Iterative Version

Algorithm *downHeap*(*H, size*) // When delete
Input Array H representing a min-heap and the size of H (size ≥ 1)
Output H with the heap property restored
property ← **false // Heap Property boolean variable**
i ← **1** // start with the heap root
while ! ***property*** **do**
/* Find the index of the smallest element between the current node and its children using the indexOfMin function*/
smallest ← **indexOfMin(H, i, size)**
if ***smallest*** **≠ i then** // smallest is not the current swap
swapElements(*H*[*smallest*], *H*[*i*])
i ← **smallest** // Update i to the index of the smallest element and repeat the process
else
property ← true
return H

Helper method to find the min-index

Algorithm *indexOfMin*(*A, r, size*)
Input array*A*, an index *r* (referencing an item of A), and *size* of the heap stored in *A*
Output the rank of element in *A* containing the smallest value
smallest ← **r** // referring the current node

// Need to know the left and right child position to check the smallest index.
left ← **2*r** // Array based heap left child formula
right ← **2*r + 1.** // Array based heap right child formula

// If left is within the array size and the left child is less than the current value assign left to the smallest
if left ≤ size and key(*A*[left]) < key(*A*[smallest]) then // Smallest indicate the parent
smallest ← **left**

// If right is within the array size and the right child is less than the current value assign right to the smallest
if right ≤ size and key(*A*[right]) < key(*A*[smallest]) then
smallest ← **right**
return *smallest*

Bottom-up heap construction – Recursive Algorithm (Not for the exam) -Just for the knowledge

Algorithm BottomUpHeap(A)
Input: An array A storing n = 2^h-1 keys
Output: A heap T storing the keys in A

if A is empty **then**
return an empty heap
remove the first key *k* from A
split A into two subarrays, A_1 and A_2, of equal length
$T_1 \leftarrow$ BottomUpHeap(A_1)
$T_2 \leftarrow$ BottomUpHeap(A_2)
create a binary tree T with root r storing k, left subtree T_1, right subtree T_2
perform a downheap from the root r of T, if necessary
return T

NOTE: The algorithm works without modification when n has the form 2^h-1; it needs to be modified slightly for other n (but the results concerning its running time still hold true).

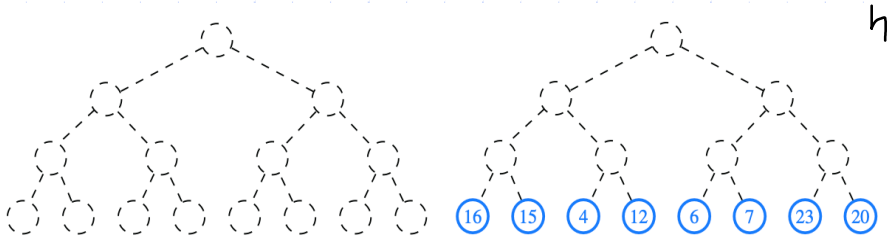
BottomUpHeap Iteratively

- Array size n is of the form 2^h-1, Steps to follow
  - In the first step, we construct $(n + 1)/2$ elementary heaps storing one entry each. The remining (n-1)/2 items still in reserve.
  - In the second step, we form$(n+1)/4$ heaps, by joining pairs of elementary heaps and adding a new entry. The new entry is placed at the root and may have to be swapped with the entry stored at a child to preserve the heap-order property.
  - In the third step, we form $(n + 1)/8$ heaps, by joining pairs 3 entry heaps constructed in the previous step and adding a new entry. The new entry is placed initially at the root but may have to move down with a down-heap bubbling to preserve the heap-order property.
  - Repeat log(n+1) times (= height of the tree)

BottomUpHeap Construction- Iterative

Input Array [16, 15, 4, 12, 6, 7, 23, 20, 25, 9, 11, 17, 5, 8, 14]

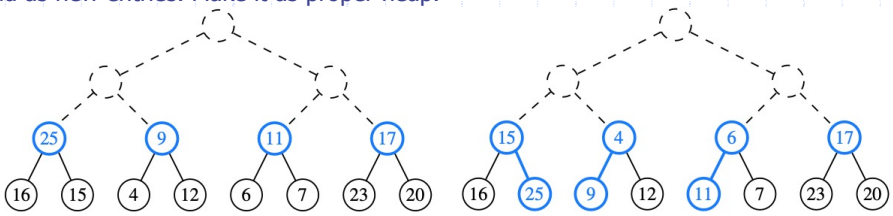
Step 1: N=15 **N+1/2 = 8**, first eight elements as leaf entries



(a) (b)

Step 2: N=15

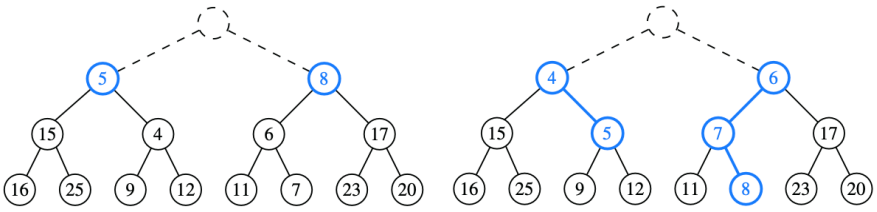
N+1/4 = 4, take the next 4 elements, joining the pairs of each elements as a root and add as new entries. Make it as proper heap.



(c) (d)

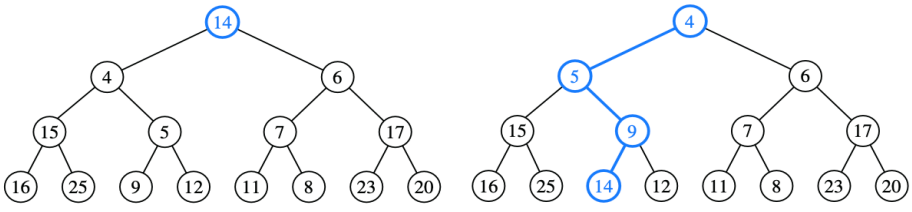
BottomUpHeap Construction- Iterative

Step 3: N=15 N+1/8 = 2, Take the next two elements. Each side storing 7 entries, by joining the pairs of three entries. Make it as a proper heap.



(e) (f)

Last Step, take the last element and make it as a root by jointing the previous step entries. Make it as a Proper heap.



(g) (h)

Running Time

- Both the recursive and iterative versions of Bottom Up Heap run in O(n).
- With small (but messy) modifications, both the recursive and iterative versions can be used on any input size (not just a power of two minus 1) – these modified algorithms also run in O(n) time.
- A simplified Bottom Up Heap algorithm applied to input of size 2^h-1 generates a completely filled binary tree (in O(n) time), as described in Lesson 5. (In the simplified version, there is no concern about maintaining the Heap Order Property.)

When I can go for Top Down and Bottomup

Top down: If you don't know the input size
Bottom Up: If you know how many inputs are there and size = 2^h – 1.

Review Questions

1. Mention the heap ADT operations and its performance.
2. What is heap and its properties? (Slide-6)
3. Able to perform insertion and deletion heap operation using upheap and down heap.
4. How to represent a heap using Array?