Heap

Ja-Hee Kim

Agenda

01 ADT of Priority Queue

ADT for searching a node with the highest priority

02 Heap

Data structure implementing a priority queue

03 Operations of a heap

Insert, removeMin

ADT of a Priority Queue

ADT: Priority Queue

- A priority queue is the abstract data type (ADT) of an ordered set with the operations:
    - find element with highest priority
    - delete element with highest priority
    - insert element with assigned priority

- Every element should be comparable.

Operations of a Priority Queue

- add:
    - Task: add a given element to the priority queue.
    - Input: newEntry is a new entry.
- remove:
    - Task: if the priority queue is not empty, removes and returns the entry having the highest priority
    - Output: Returns either the object having the highest priority or if the priority queue is empty before the operation, null.
- peek
    - Task: Retrieves the entry having the highest priority.
    - Output: Returns either the entry having the highest priority or null if the priority queue is empty.
- isEmpty:
    - Task: detects whether it is empty.
    - Output: returns true if it is empty or false otherwise.
- clear:
    - Task: removes all entries from the priority queue

PriorityQueue<T>
+add(T newEntry):void +remove():T +peek():T +isEmpty():Boolean +clear():void

BuiltIn class: Priority Queue

- Included in java.util package

```java
import java.util.*;
public class TestPQ {
    public static void main(String[] args) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        pq.add(2);
        pq.add(9);
        pq.add(7);
        pq.add(6);
        pq.add(5);
        pq.add(8);
        pq.add(10);
        System.out.println("Remove the data with the highest priority: "+ pq.remove());
        System.out.println("Peek the data with the next priority: "+ pq.peek());
    }
}
```

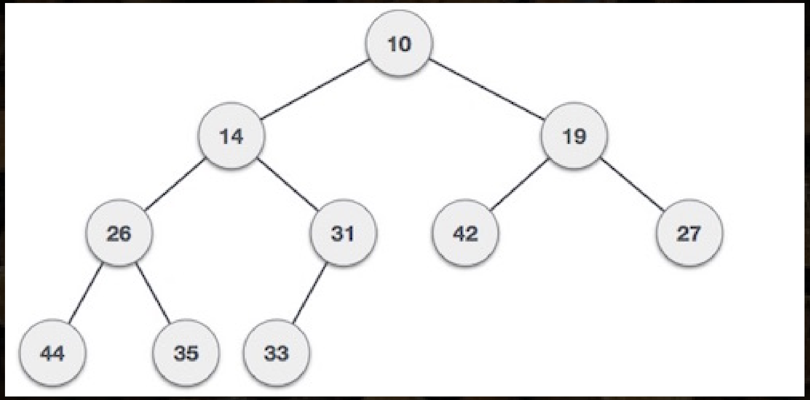
Problems @ Javadoc Declaration Console

```
<terminated> TestPQ [Java Application] C:₩Program Files₩Java₩jdk-14.0.2₩bin₩javaw.exe (2020. 10. 28. 오전 3:21:31 – 오전 3:21:31)
Remove the data with the highest priority: 2
Peek the data with the next priority: 5
```

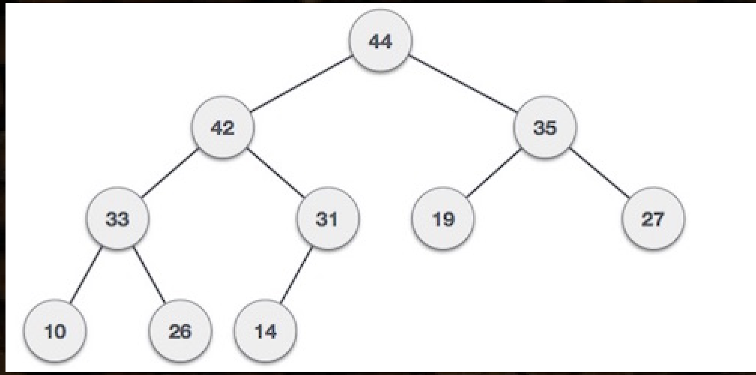
Heap

Implementation of a Priority Queue: Heap

- Heaps are very good for implementing priority queues
- Heap
  - Complete binary tree with keys
  - It satisfies two properties:
    - MinHeap: key(parent) <= key(child)
    - [OR MaxHeap: key(parent) >= key(child)]



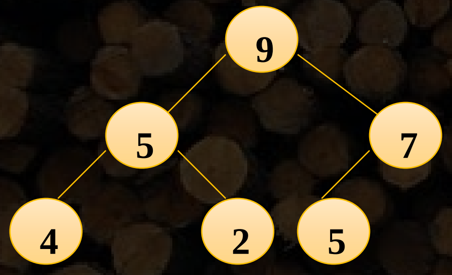
Min-Heap – Where the value of the root node is less than or equal to either of its children.



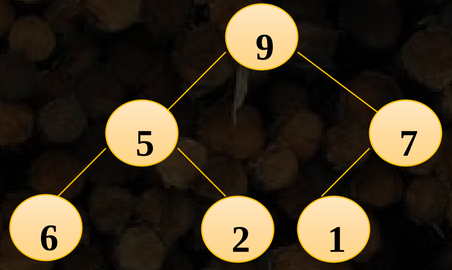
Min-Heap – Where the value of the root node is less than or equal to either of its children.

Heap visualization: https://www.cs.usfca.edu/~galles/visualization/Heap.html

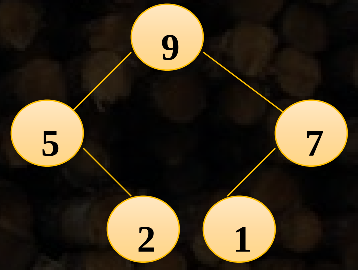
Question: Heap or not?



tree 1



tree 2

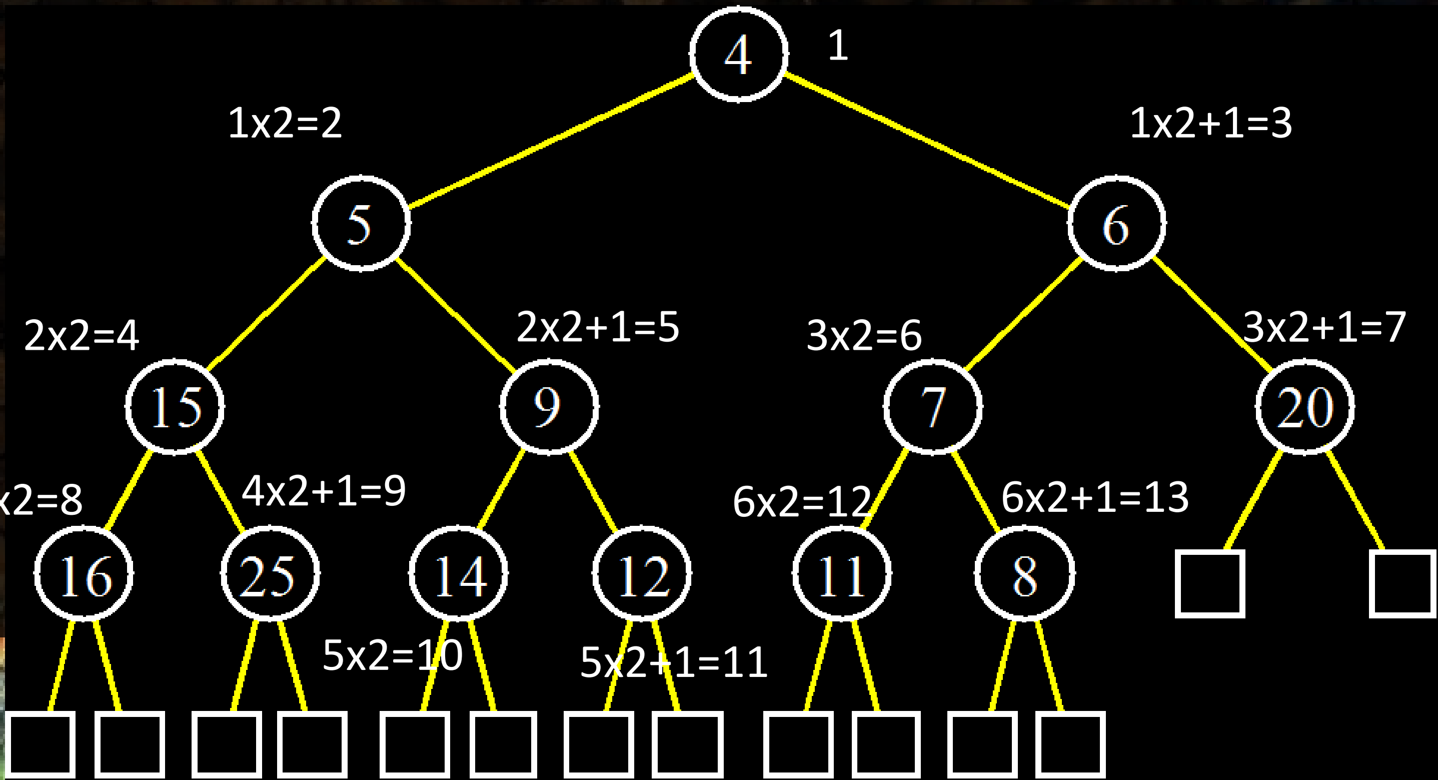


tree 3

Implementation of a Heap

- A heap is a complete binary tree, so it is implemented using an array.
- If you know the index of a node, then it is easy to figure out the indexes of that node's parent and children.

	4	5	6	15	9	7	20	16	25	14	12	11	8
0	1	2	3	4	5	6	7	8	9	10	11	12	13



Operations of a Heap

Insertion

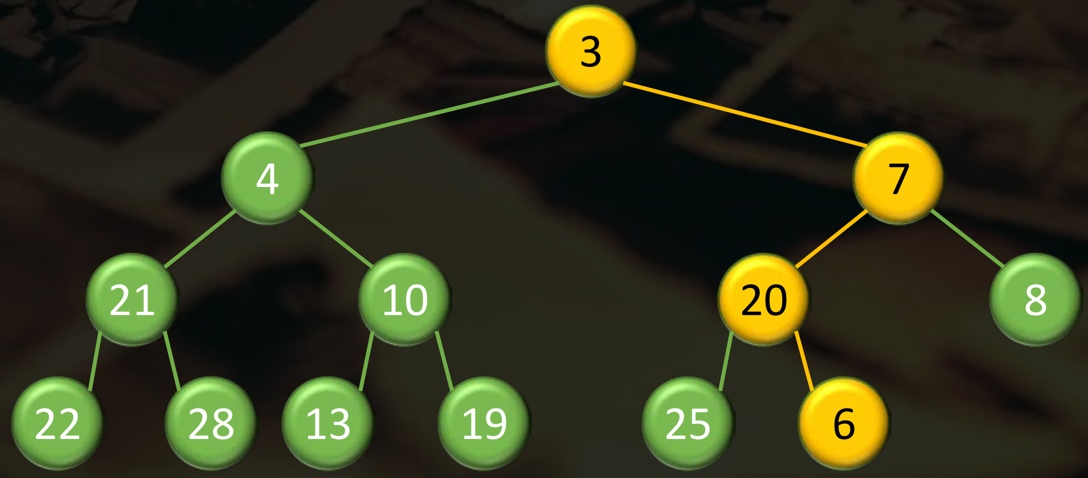
1. A Add key in next available position.
2. Begin upheap (swap with its parent)
3. Terminate upheap when (reach root/key child is greater than key parent)

Example: insert 6

1. Add key in next available position.
2. Begin upheap (swap with its parent)
3. Terminate upheap when (reach root/key child is greater than key parent)

Example: insert 6

1. Add key in next available position.
2. Begin upheap (swap with its parent)
3. Terminate upheap when (reach root/key child is greater than key parent)

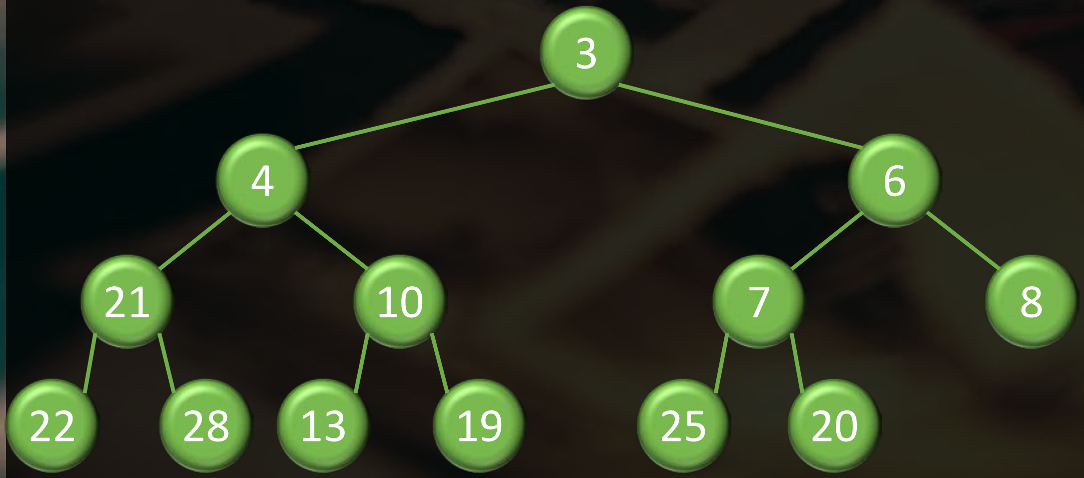


Deletion

1. Return the root and delete it from the heap.
2. Move the last one to the root.
3. Begin downheap (swapping the smaller child).
4. Terminate downheap when (reach leaf level/key parent is greater than key child)

Deletion

1. Return the root and delete it from the heap.
2. Move the last one to the root.
3. Begin downheap (swapping the smaller child).
4. Terminate downheap when (reach leaf level/key parent is greater than key child)

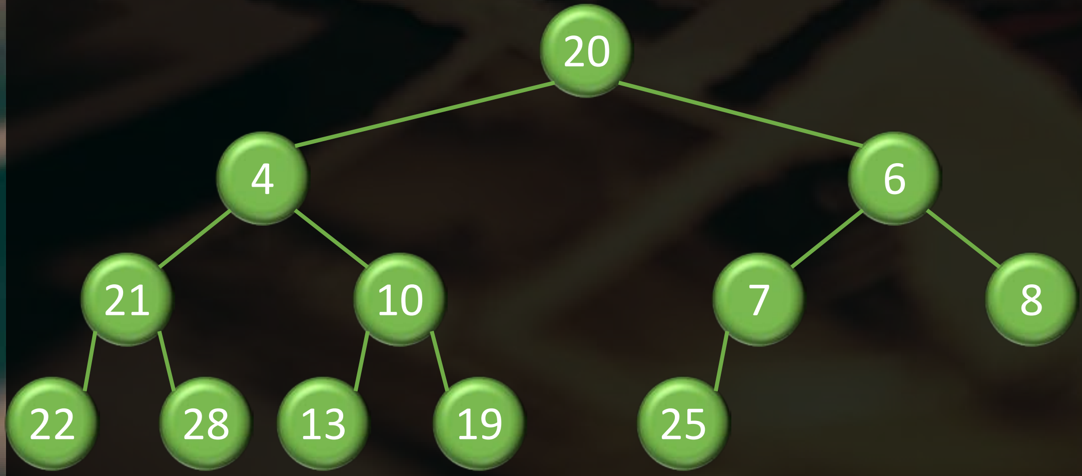


Deletion

1. Return the root and delete it from the heap.
2. Move the last one to the root.
3. Begin downheap (swapping the smaller child).
4. Terminate downheap when (reach leaf level/key parent is greater than key child)

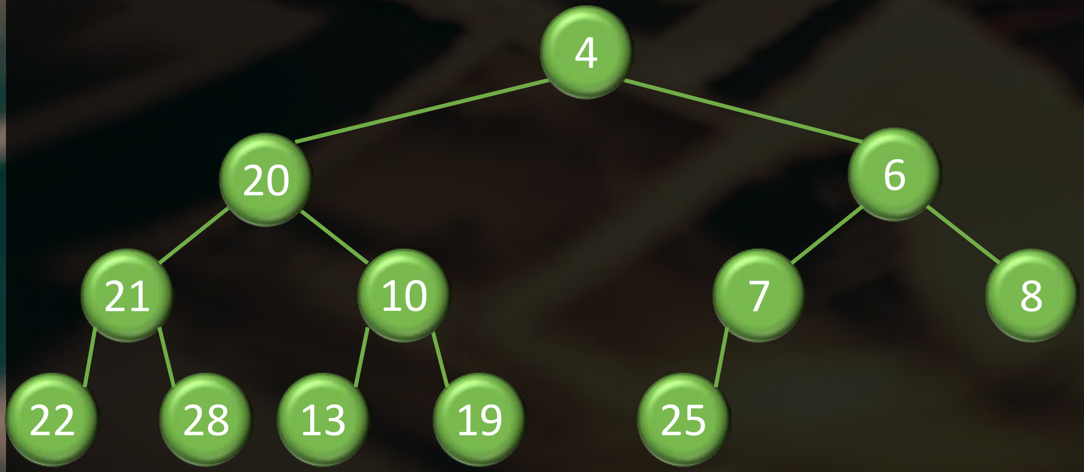
Deletion

1. Return the root and delete it from the heap.
2. Move the last one to the root.
3. Begin downheap (swapping the smaller child).
4. Terminate downheap when (reach leaf level/key parent is greater than key child)



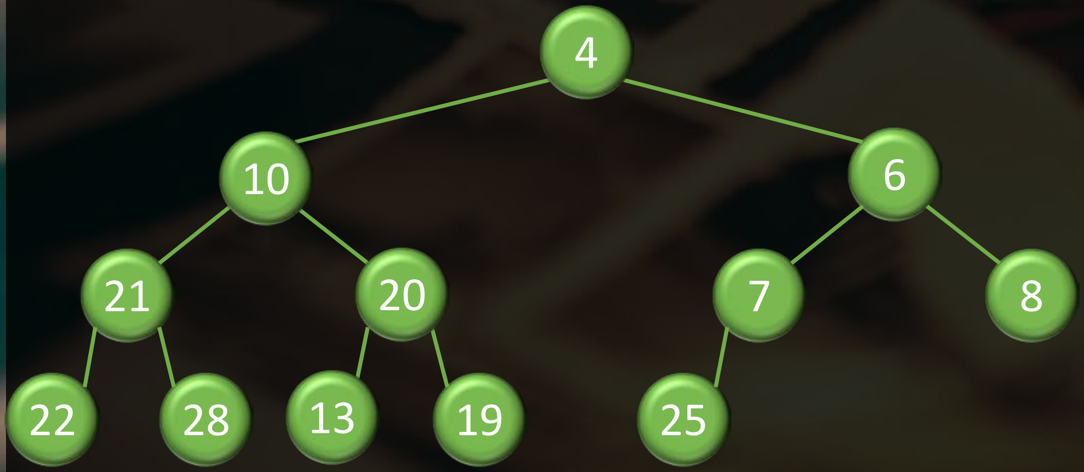
Deletion

1. Return the root and delete it from the heap.
2. Move the last one to the root.
3. Begin downheap (swapping the smaller child).
4. Terminate downheap when (reach leaf level/key parent is greater than key child)



Deletion

1. Return the root and delete it from the heap.
2. Move the last one to the root.
3. Begin downheap (swapping the smaller child).
4. Terminate downheap when (reach leaf level/key parent is greater than key child)



Construction of a Heap

Bottom-up vs. Top-down heap construction

- **Bottom-up**: Put everything in and then fix it

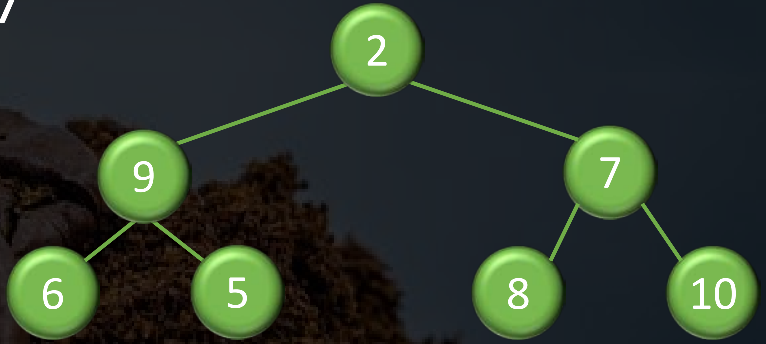
- **Top down**: Heaps can be constructed by successively inserting elements into an (initially) empty heap

Bottom-up construction

- Insert elements in the order given breadth-first in a binary tree
- Starting with the last (rightmost) parental node, fix the heap rooted at it, if it does not satisfy the heap condition:
  - exchange it with its smallest child
  - fix the subtree rooted at it (now in the child’s position)
- Example: 2 9 7 6 5 8 10
- Efficiency:

Example: 2 9 7 6 5 8 10

	2	9	7	6	5	8	10
0	1	2	3	4	5	6	7

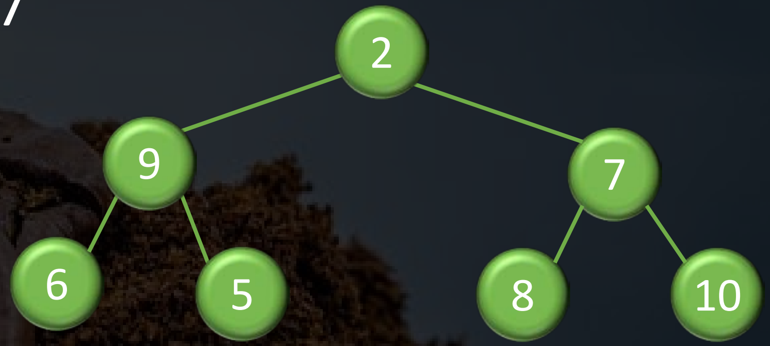


Starting with **the last (rightmost) parental node**, fix the heap rooted at it, if it does not satisfy the heap condition:

- exchange it with its smallest child
- fix the subtree rooted at it (now in the child's position)

Example: 2 9 7 6 5 8 10

	2	9	7	6	5	8	10
0	1	2	3	4	5	6	7

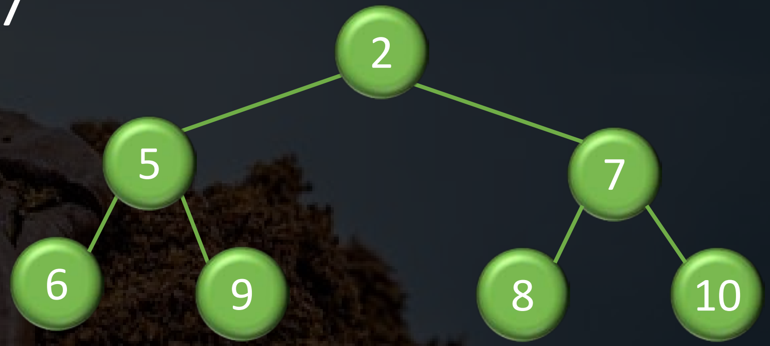


Starting with the last (rightmost) parental node, fix the heap rooted at it, if it does not satisfy the heap condition:

- exchange it with its smallest child
- fix the subtree rooted at it (now in the child's position)

Example: 2 9 7 6 5 8 10

	2	5	7	6	9	8	10
0	1	2	3	4	5	6	7



Starting with the last (rightmost) parental node, fix the heap rooted at it, if it does not satisfy the heap condition:

- exchange it with its smallest child
- fix the subtree rooted at it (now in the child's position)

Efficiency of bottom-up construction

- For parental node at level *i* it does 2(*h*-*i*) comparisons in the worst case

- Total:

$$\sum_{i=0}^{h-1} 2(h-i)\, 2^{i} \;=\; 2\,(\,n - \lg\,(n+1)) \;=\; \Theta(n)$$

nodes at level *i*

Top-down heap construction

- Heaps can be constructed by successively inserting elements into an (initially) empty heap

- Insert element at last position in heap
- Compare with its parent and if it violates heap condition exchange them
- Continue comparing the new element with nodes up the tree until the heap condition is satisfied

Example: 2 9 7 6 5 8 10

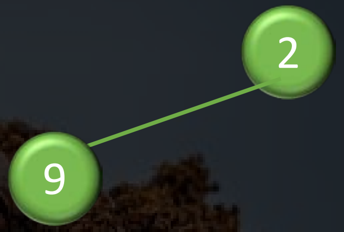
	2	9	7	6	5	8	10
0	1	2	3	4	5	6	7

2

1. Insert element at last position in heap
2. Compare with its parent and if it violates heap condition exchange them
3. Continue comparing the new element with nodes up the tree until the heap condition is satisfied

Example: 2 9 7 6 5 8 10

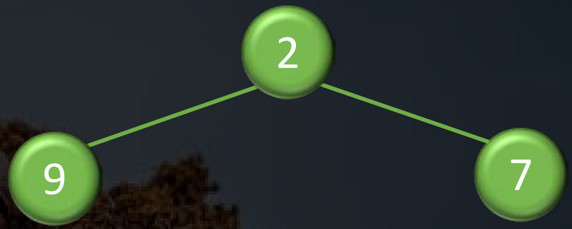
	2	9	7	6	5	8	10
0	1	2	3	4	5	6	7



1. Insert element at last position in heap
2. Compare with its parent and if it violates heap condition exchange them
3. Continue comparing the new element with nodes up the tree until the heap condition is satisfied

Example: 2 9 7 6 5 8 10

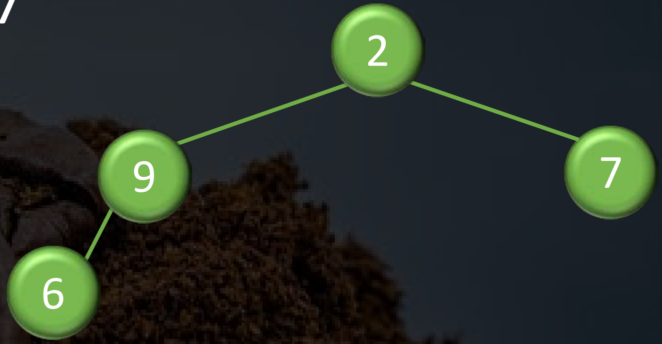
	2	9	7	6	5	8	10
0	1	2	3	4	5	6	7



1. Insert element at last position in heap
2. Compare with its parent and if it violates heap condition exchange them
3. Continue comparing the new element with nodes up the tree until the heap condition is satisfied

Example: 2 9 7 6 5 8 10

	2	9	7	6	5	8	10
0	1	2	3	4	5	6	7



1. Insert element at last position in heap
2. Compare with its parent and if it violates heap condition exchange them
3. Continue comparing the new element with nodes up the tree until the heap condition is satisfied

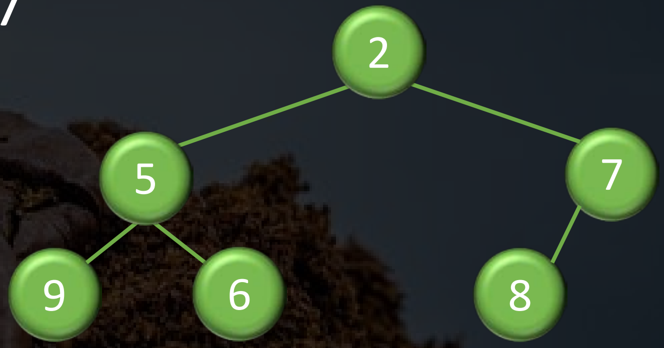
Example: 2 9 7 6 5 8 10

	2	6	7	9	5	8	10
0	1	2	3	4	5	6	7

1. Insert element at last position in heap
2. Compare with its parent and if it violates heap condition exchange them
3. Continue comparing the new element with nodes up the tree until the heap condition is satisfied

Example: 2 9 7 6 5 8 10

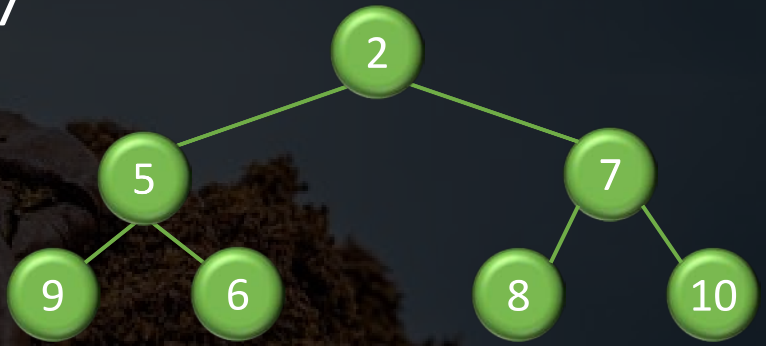
	2	5	7	9	6	8	10
0	1	2	3	4	5	6	7



1. Insert element at last position in heap
2. Compare with its parent and if it violates heap condition exchange them
3. Continue comparing the new element with nodes up the tree until the heap condition is satisfied

Example: 2 9 7 6 5 8 10

	2	5	7	9	6	8	10
0	1	2	3	4	5	6	7



1. Insert element at last position in heap
2. Compare with its parent and if it violates heap condition exchange them
3. Continue comparing the new element with nodes up the tree until the heap condition is satisfied

Efficiency

- Insert element at last position in heap.
- Compare with its parent and if it violates heap condition exchange them
- Continue comparing the new element with nodes up the tree until the heap condition is satisfied

Efficiency: $\Sigma \log(i) \approx n \log(n)$

Thank you