The heap data structure

1. Heap background
2. Heap algorithms
3. Runtime efficiencies

1. Heap background

A heap is a binary tree with certain properties.

Assumption: each vertex (node) in the binary tree has a value (a key) associated with it

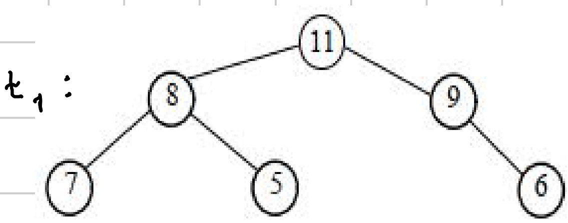
To be a heap a binary tree must have the following properties:

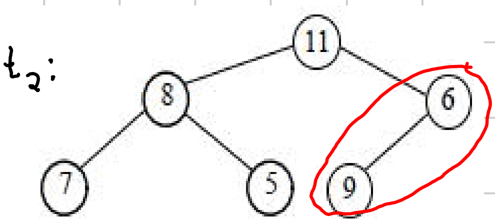
property 1: the parent's key is at least as large as the keys of its children

property 2: all depths, except possibly the largest, have the maximum number of nodes

property 3: at the largest depth, any missing nodes (leaves) are at the right end

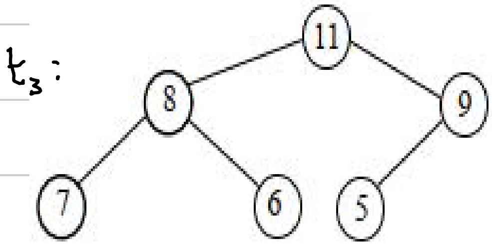
Example Is the binary tree a heap?



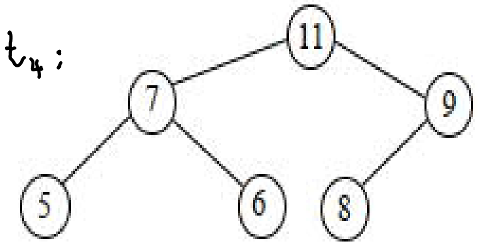


No. Does not have property 3.

No. Does not have property 1.



Yes.



Yes.

□

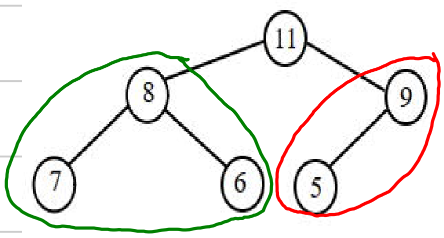
Notes

1. Property 1 is required for a max-heap. In a min-heap, the parent's key is at most as large as the keys of its children.

2. Given a set of keys there may be several binary trees that satisfy the heap properties. Hence a heap is not unique.

3. For each node, both its right subtree and its left subtree are heaps. (Hence: suitable for recursion).

4. In a max-heap, the maximum key is always at the root.



Both subtrees are heaps.

Heap parameters

$$n = \text{number of nodes in heap}$$

$$n_{leaf} = \text{number of leaves in a heap}$$

$$h = \text{height of heap}$$

Relationships:

$$2^h \le n < 2^{h+1}$$

$$h = \lfloor \log_2(n) \rfloor$$

$$n_{leaf} = \lceil n/2 \rceil$$

A priority queue is often implemented using a heap.

Priority queue:

- a collection of items each having a priority
- allows easy access to the item with the highest priority
- allows the item with the highest priority to be removed
- allows new item to be added to

2. Heap algorithms

Assume the following are always available:

node.parent = pointer to parent of node or NIL if node is *heapRoot*

node.left = pointer to left child of node or NIL if there is no left child

node.right = pointer to right child of node or NIL if there is no right child *node.key* = key value associated with node

heapRoot = root node of heap (represents entire heap)

height = the height of the heap

Four algorithms:

HEAP-INSERT: allows insertion of a new node into an existing heap

HEAPIFY: for a binary tree (or subtree) lacking property 1 at the root, restores this property thereby making a heap

BUILD-HEAP: for a binary tree lacking property 1 at any (and possibly all) nodes, restores this property thereby making a heap

HEAP-EXTRACT-MAX: removes the root from a heap, restores the heap properties and returns the maximum key

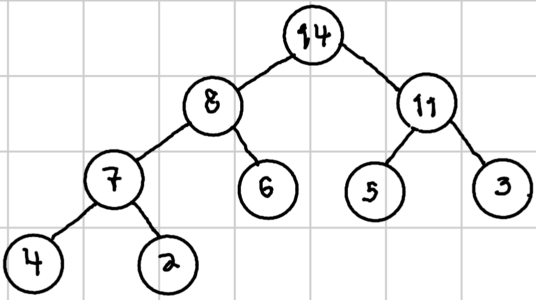
HEAP-INSERT-pseudocode

```
HEAP–INSERT(heapRoot, node)
input a heap whose root node is specified by heapRoot and a
new node that should be added to the heap

if depth height is not yet full of leaves then
  ▷ insert node as the rightmost leaf at depth height
else
  height = height + 1
  ▷ insert node as the first (leftmost) leaf at depth height
end

/* After this insertion, node now has a parent. We allow
node to percolate up in the heap until it finds its
correct location. */
while (node ≠ heapRoot and node.key > node.parent.key)
   SWAP(node, node.parent)
end
```

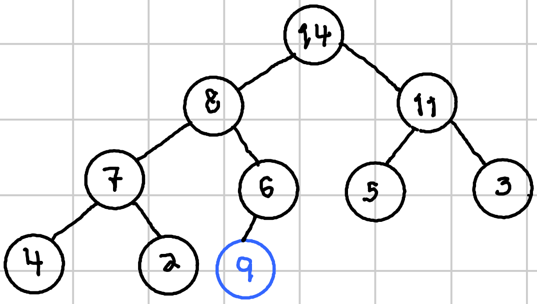
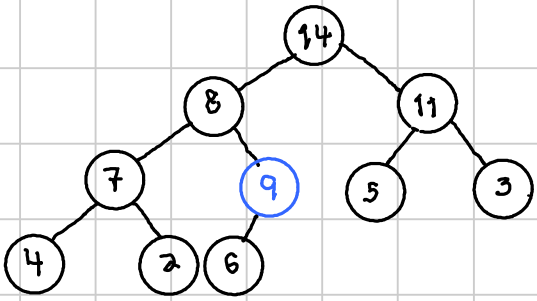
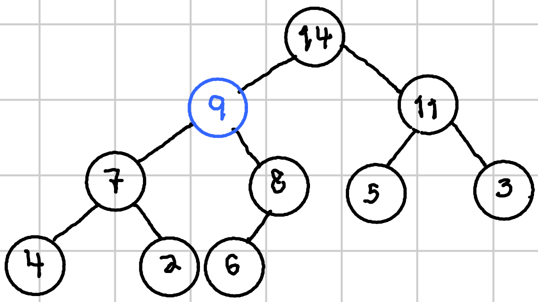
NOTE: In a SWAP a node retains its key, but its pointers are updated.

Example

heap at start:



node to add 9

stage	line	new heap
1	6	
2	16	
3	16	

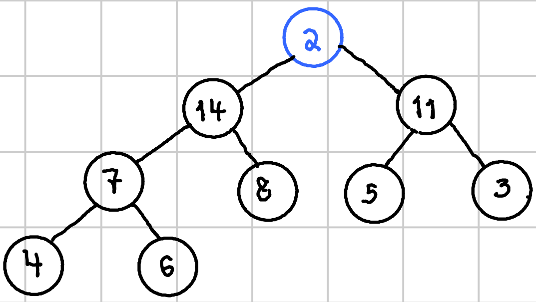
□

HEAPIFY-pseudocode

```
1   HEAPIFY(heapRoot, node1)
2   input a heap whose root node is specified by heapRoot and
3   one particular node (node1) of the heap
4   /* All nodes in the heap, except for possibly node1,
5   have the heap properties. We allow node1 to trickle down to
6   its correct location. */
7
8   node2 = NIL
9   while (node1 ≠ node2)
10    node2 = node1
11    L = node1.left, R = node1.right
12    if (L exists and L.key > node2.key) then
13       node2 = L
14    end
15    if (R exists and R.key > node2.key) then
16       node2 = R
17    end
18    if (node1 ≠ node2) then
19      SWAP(node1, node2)
20    end
21  end
```

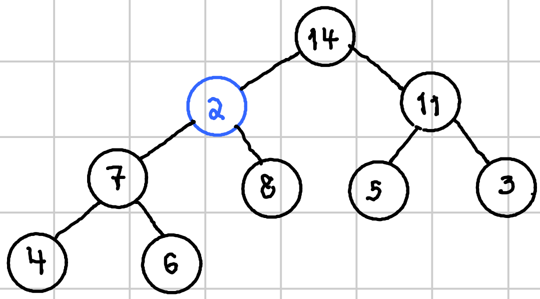
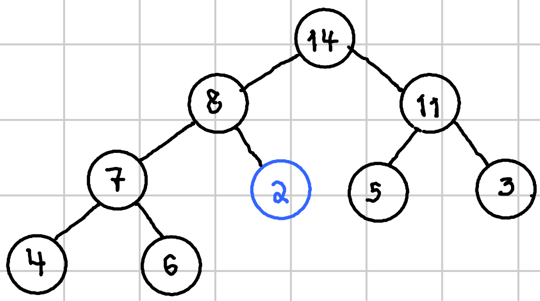
Example

starting binary tree



node1: 2

stage	line	computation or new heap
1	10	node2 = 2
2	11	L = 14 R = 11
3	13	node2 = 14

4	19	
5	10, 11	node2 = 2, L = 7 R = 8
6	16	node2 = 8
7	19	
8	10, 11	node2 = 2, L = NIL, R = NIL

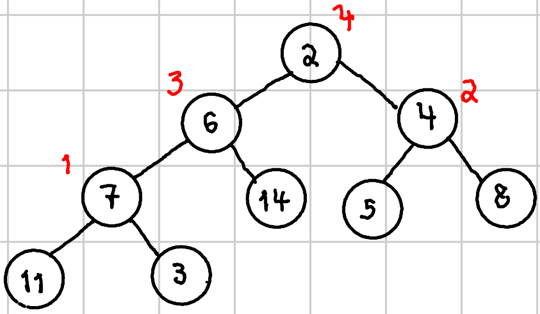
□

BUILD-HEAP-pseudocode

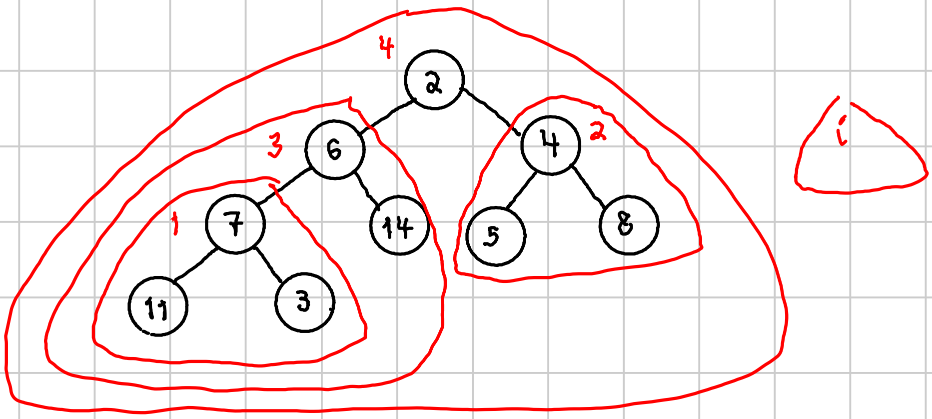
```
1   BUILDHEAP(heapRoot)
2   input a binary tree whose root node is specified by heapRoot
3   /* The input binary tree may lack property 1 at any or all
4   of its nodes. As such all nodes, except for the current
5   leaves, must be checked and if necessary modified so that
6   the final output is a binary tree which is a heap. */
7
8   for i from height − 1 to 0
9     forEach internal node at depth i from right to left
10      HEAPIFY(heapRoot, node)
11    end
12  end
```

Example

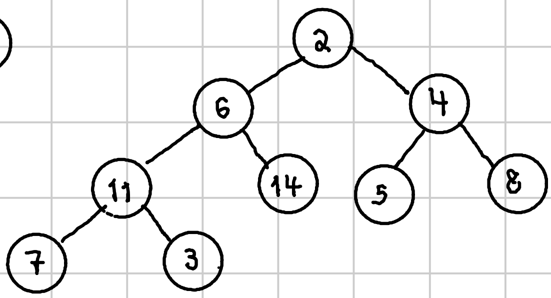
starting binary tree



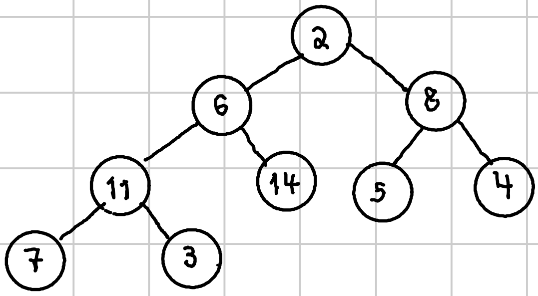
i = order nodes are handled in



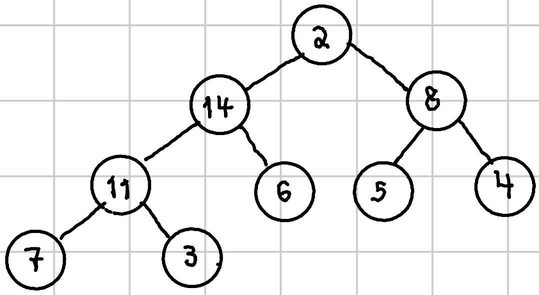
i order subheaps are processed

stage	line		new binary tree
1	10	node = 7	

2 10 node = 4



3 10 node = 6



4 10 node = 2

?

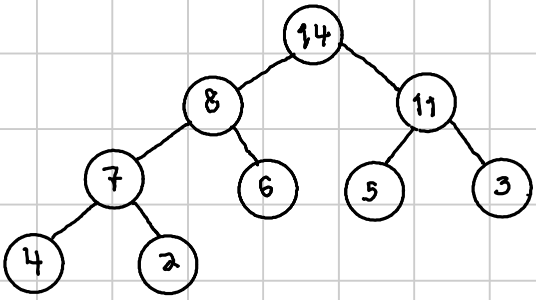
□

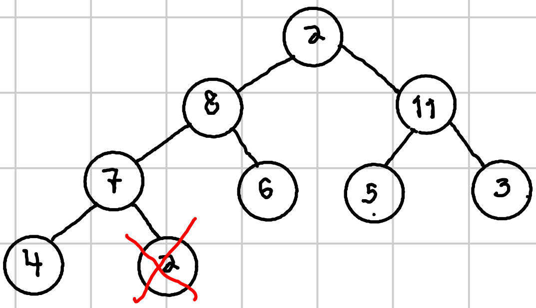
HEAP-EXTRACT-MAX-pseudocode

```
HEAP–EXTRACT–MAX(heapRoot)
input a heap whose root node is specified by heapRoot
/* The root of the heap is removed and replaced by the
rightmost leaf at the lowest depth. After restoring the
heap properties, the key of the original root is returned. */

max = heapRoot.key
▷ let node be the rightmost leaf at depth height
heapRoot.key = node.key
▷ remove node from the heap
if node was the only node at depth height
  height = height − 1
end
HEAPIFY(heapRoot, heapRoot)
return max
```

Example

heap at start:



stage	line	new heap or computation
1	7, 8	max = 14, node = 2
2	9, 10	
3	14	?

□

3. Runtime efficiencies

What are the runtime efficiencies of HEAP-INSERT, HEAPIFY, BUILD-HEAP and HEAP-EXTRACT-MAX?

HEAP-INSERT efficiency

Q: How many iterations in **while**-loop?

```
HEAP–INSERT(heapRoot, node)
input a heap whose root node is specified by heapRoot and a
new node that should be added to the heap

if depth height is not yet full of leaves then
  ▷ insert node as the rightmost leaf at depth height
else
  height = height + 1
  ▷ insert node as the first (leftmost) leaf at depth height
end

/* After this insertion, node now has a parent. We allow
node to percolate up in the heap until it finds its
correct location. */
while (node ≠ heapRoot and node.key > node.parent.key)
   SWAP(node, node.parent)
end
```

A: At most one more than height of the starting heap (so *h+1*).

runtime efficiency of HEAP-INSERT: $O(h) = O(\log_2 n)$

HEAPIFY efficiency

Q: How many iterations in **while**-loop?

```
HEAPIFY(heapRoot, node1)
input a heap whose root node is specified by heapRoot and
one particular node (node1) of the heap
/* All nodes in the heap, except for possibly node1,
have the heap properties. We allow node1 to trickle down to
its correct location. */

node2 = NIL
while (node1 ≠ node2)
  node2 = node1
  L = node1.left, R = node1.right
  if (L exists and L.key > node2.key) then
      node2 = L
  end
  if (R exists and R.key > node2.key) then
      node2 = R
  end
  if (node1 ≠ node2) then
    SWAP(node1, node2)
  end
end
```

A: At most height of the starting heap (so *h*).

runtime efficiency of HEAPIFY: $O(h) = O(\log_2 n)$

HEAP-EXTRACT-MAX efficiency

Q: What is the runtime efficiency of HEAP-EXTRACT-MAX?

A: The same as that of HEAPIFY.

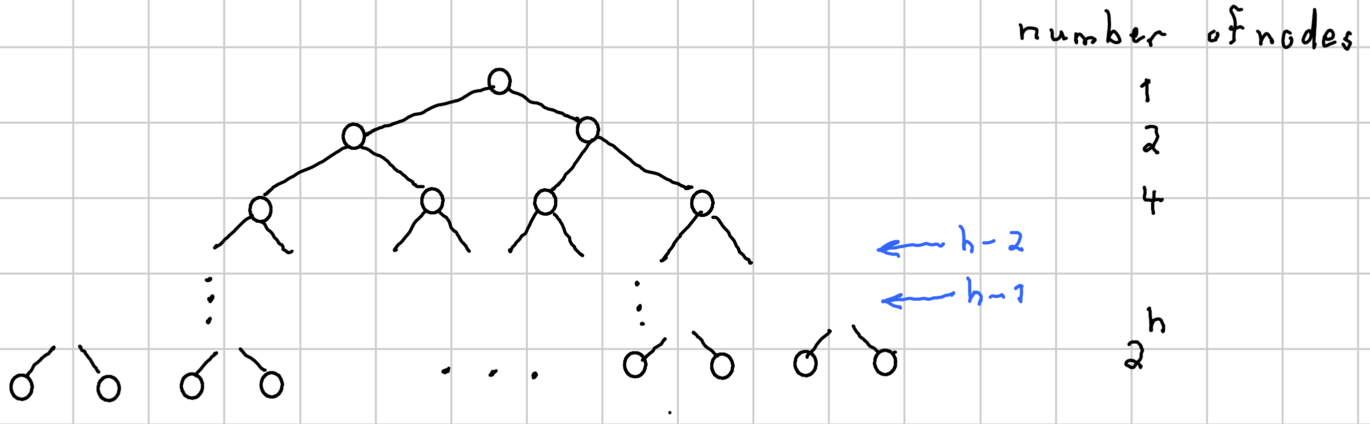
```
HEAP-EXTRACT-MAX(heapRoot)
input a heap whose root node is specified by heapRoot
/* The root of the heap is removed and replaced by the
rightmost leaf at the lowest depth. After restoring the
heap properties, the key of the original root is returned. */

max = heapRoot.key
▷ let node be the rightmost leaf at depth height
heapRoot.key = node.key
▷ remove node from the heap
if node was the only node at depth height
  height = height − 1
end
HEAPIFY(heapRoot)
return max
```

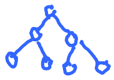
runtime efficiency of HEAP-EXTRACT-MAX: $O(h) = O(\log_2 n)$

BUILDHEAP efficiency

Assumption: starting heap is a complete binary tree, so $n = 2^{h+1} - 1$.

```
BUILDHEAP(heapRoot)
input a binary tree whose root node is specified by heapRoot
/* The input binary tree may lack property 1 at any or all
of its nodes. As such all nodes, except for the current
leaves, must be checked and if necessary modified so that
the final output is a binary tree which is a heap. */

for i from height − 1 to 0
  forEach internal node at depth i from right to left
    HEAPIFY(heapRoot, node)
  end
end
```



depth	number of nodes	simple operations in a single HEAPIFY	total simple operations
h	$(n+1)/2$	0 (no calls to HEAPIFY)	0
$h-1$	$(n+1)/4$	 1	$1 \cdot (n+1)/4$
$h-2$	$(n+1)/8$	 2	$2 \cdot (n+1)/8$
$h-3$	$(n+1)/16$	3	$3 \cdot (n+1)/16$
$\vdots$	$\vdots$		
0	$(n+1)/2^h$ $(=1)$	h	$h \cdot (n+1)/2^h$

Sum of all simple operations:

$$\frac{n+1}{4} + \frac{2(n+1)}{8} + \frac{3(n+1)}{16} + \dots \frac{h(n+1)}{2^h}$$

$$= (n+1)\sum_{i=1}^{h} \frac{i}{2^{i+1}}$$

$$\leq 2(n+1)\sum_{i=1}^{h} \frac{i}{2^{i+1}} = (n+1)\sum_{i=1}^{h} \frac{i}{2^i}$$

It can be shown: $$\sum_{i=1}^{h} \frac{i}{2^i} = 2 - \frac{h+2}{2^h} < 2$$

Conclusion: $$\frac{n+1}{4} + \frac{2(n+1)}{8} + \frac{3(n+1)}{16} + \dots \frac{h(n+1)}{2^h} \leq (n+1)\sum_{i=1}^{h} \frac{i}{2^i} < 2(n+1)$$

runtime efficiency of BUILDHEAP: $O(n)$