# Programmeren (Ectrie)

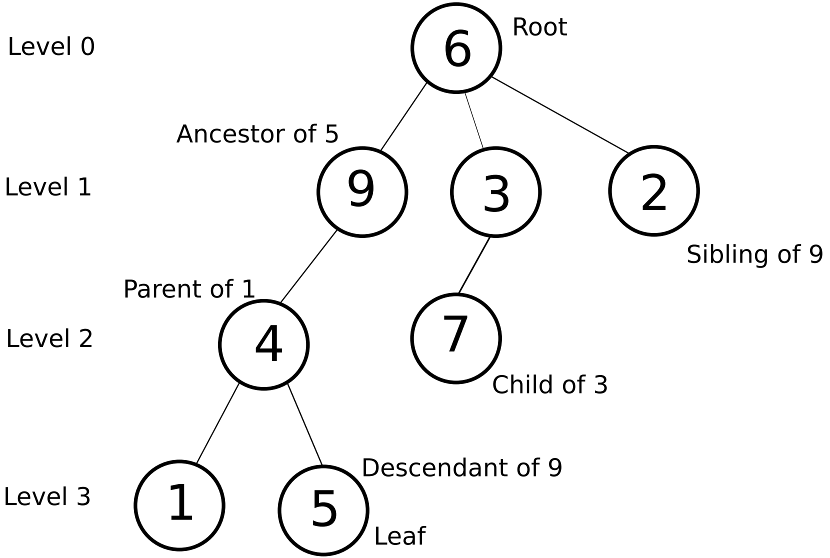
## Lecture 6: Nonlinear data structures

Tommi Tervonen

Econometric Institute, Erasmus University Rotterdam

ERASMUS UNIVERSITEIT ROTTERDAM

# Trees



Level 0
Level 1
Level 2
Level 3
6
Root
Ancestor of 5
9
3
2
Sibling of 9
Parent of 1
4
7
Child of 3
1
5
Descendant of 9
Leaf

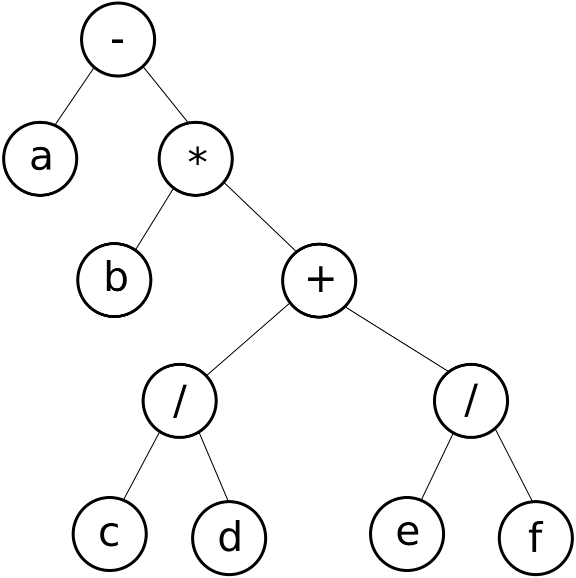
# Trees: definition and implementation in Matlab

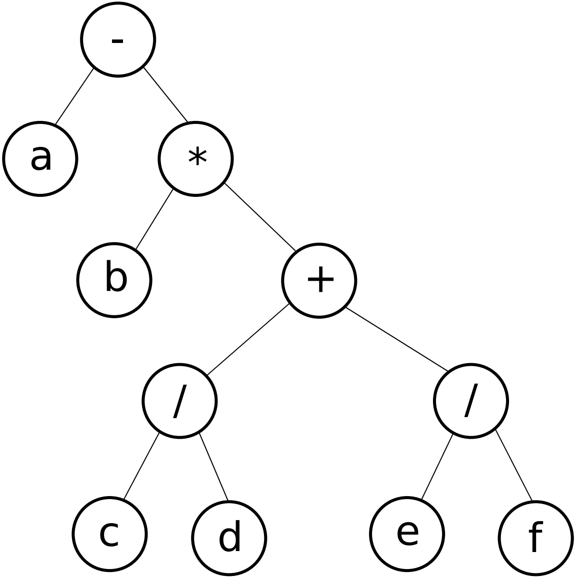
1. empty $T$ is a tree
2. if $T$ is not empty, a $T$ has exactly one node designated as the root($T$)
3. the remaining nodes $(T - root(T))$ of a tree are partitioned into $m$ disjoint sets $T_1, \ldots, T_m$. Each of these are in turn a tree, and are called subtrees of $T$.

Tree arity $m$ defines max amount of subtrees ($m = 1 \rightarrow$ linked list, $m = 2 \rightarrow$ binary tree)

```
classdef treeNode < handle
  properties
    key
    left
    right
  end
end
```

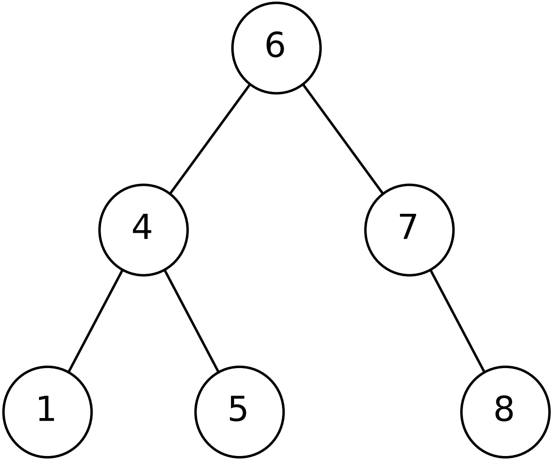
$$a - b * (c/d + e/f)$$





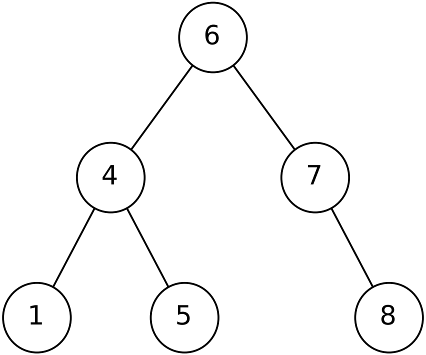
Example: tree traversal schemes; inorder, preorder and postorder

# Binary search trees (BST)

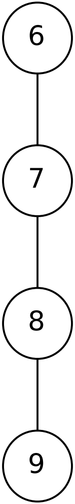


# BST search time / balanced case

- Each level: 1 comparison $\rightarrow$ 1/2 remaining nodes "discarded"
- Find complexity: $O(\log_2 n)$



# Extremely unbalanced tree

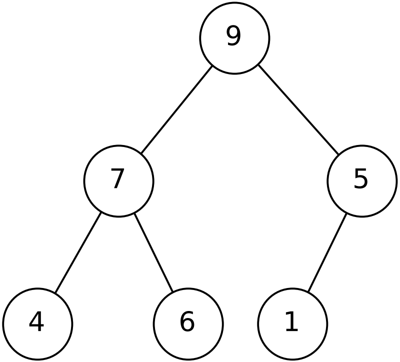


# BST operation complexity

- Insert: $O(n)$
- Delete current node: $O(1)$
- Search / balanced case: $O(\log n)$
- Search / unbalanced case: $O(n)$

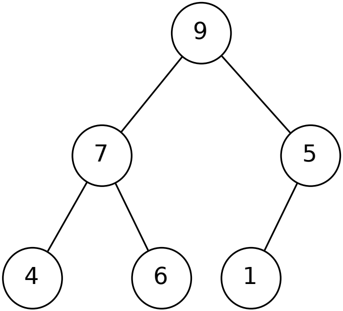
# Heap

- Balanced tree: every level of depth $x$ (except last) has exactly $2^x$ nodes
- Heap property: the key of each node is maximum that of its parent



# Heap as an array

$i^{th}$ element of $j^{th}$ level is located in the index $2^j + (i-1)$



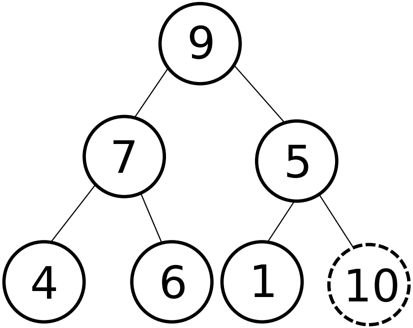

| | | | | | |
|---|---|---|---|---|---|
| 9 | 7 | 5 | 4 | 6 | 1 |

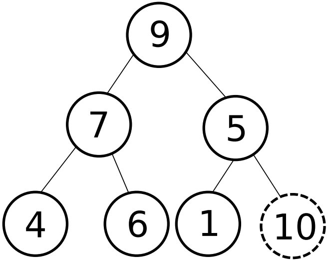
# Constructing a heap

When inserting a new node, it becomes:

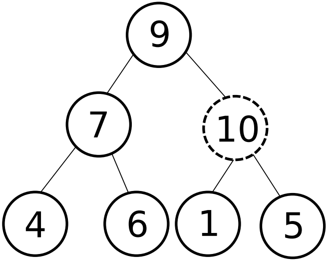
- Last node of the last layer, if there is space
- First node of a new layer (depth increases by 1)

$\Rightarrow$ possible violation of the heap property

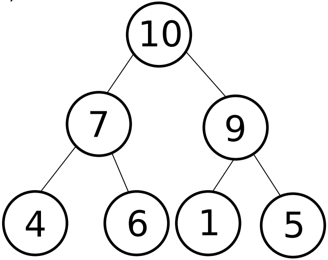





(a)




(b)




(c)

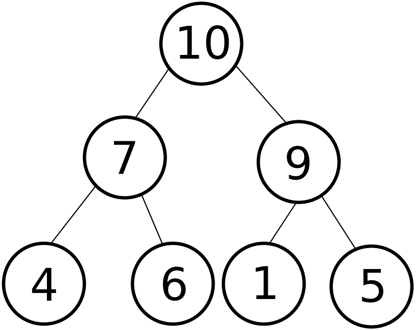
Complexity?

# Deleting from the heap

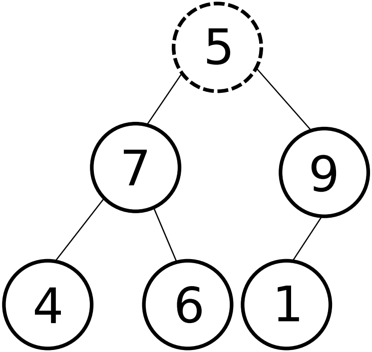
- In the heap, *only* the root node can be deleted
- Due to this, heap implements *priority queue* semantics very useful in various cases (e.g. queueing elements that some have always priority over others)
- Elegant data structure with many applications, e.g. Dijkstra's shortest path algorithm and Heapsort

# Deleting from the heap

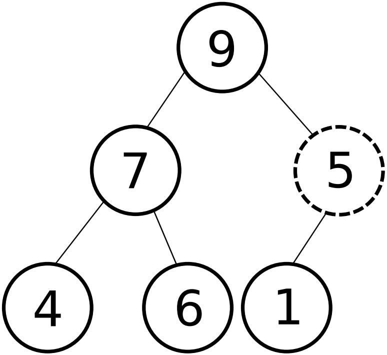
The root node is deleted and replaced with the last node
$\rightarrow$ heap balanced, but heap property violated
$\rightarrow$ heapify(root)



# heapify




(a)

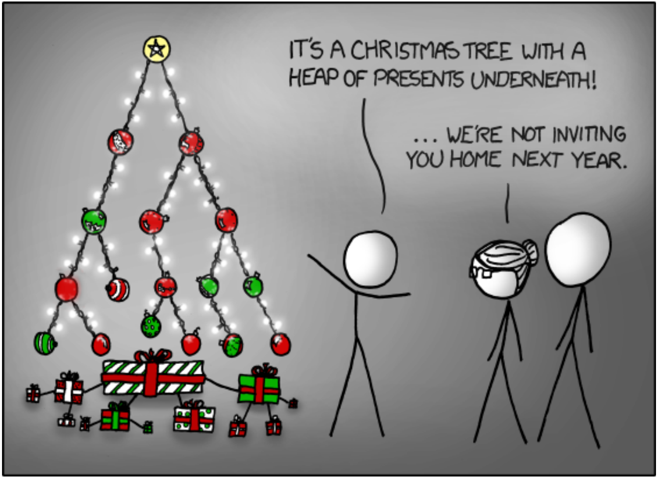



(b)

```
function H = heapify(H, n, endIndex)
  largest = 0;
  l = left(n)
  r = right(n)
  if (l <= endIndex && H(l) > H(n))
    largest = l;
  else
    largest = n;
  end
  if (r <= endIndex && H(r) > H(largest))
    largest = r;
  end
  if (largest != n)
    H = swap(H, largest, n); % pseudo-code
    H = heapify(H, largest, endIndex);
  end
end
```

# Complexity of heap operations

- Insert/delete: $O(\log n)$
- Search max: $O(1)$

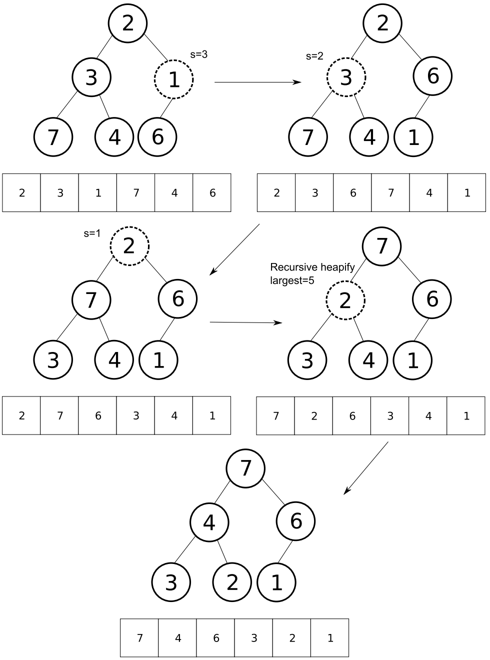


# Heapsort

Step 1: turning an arbitrary array into a heap

```
function A = buildHeap(A)
  s = floor(length(A)/2);
  while (s > 0)
    A = heapify(A, s, length(A));
    s = s - 1;
  end
end
```

Let's heapsort [2 3 1 7 4 6]



s=3
2 3 1 7 4 6
s=2
2 3 6 7 4 1
s=1
2 7 6 3 4 1
Recursive heapify
largest=5
7 2 6 3 4 1
7 4 6 3 2 1

# Complexity of buildHeap

```
function A = buildHeap(A)
  s = floor(length(A)/2);
  while (s > 0)
    A = heapify(A, s, length(A));
    s = s - 1;
  end
end
```

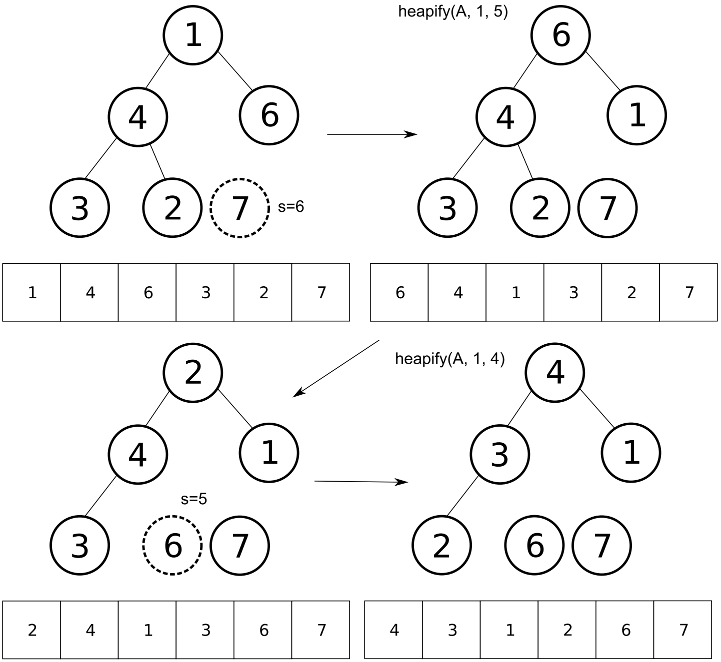
Assuming procedures (which we do not have in Matlab):

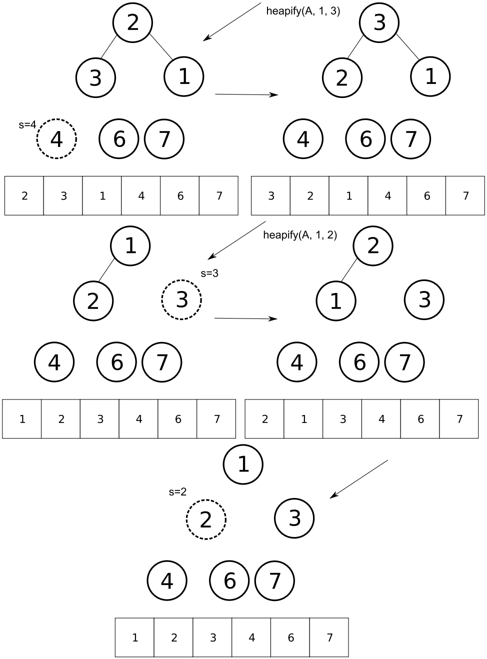
- $n/2$ iterations of while-loop
- $n$-node heap has at most $\lceil n/2^{h+1} \rceil$ nodes of height $h$
- heapify with heap of height $h$ is $O(h)$

$\Rightarrow \sum_{h=0}^{\lfloor \log_2 n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O\left(n \sum_{h=0}^{\lfloor \log_2 n \rfloor} \frac{h}{2^h}\right) = O(n2) = O(n)$

# Full heapsort

```
function A = heapSort(A)
  s = length(A);
  % until s == 2, but this is a safer condition
  while (s > 1)
    % pseudo-code
    A = swap(A, 1, s);
    A = heapify(A, 1, s);
    s = s - 1;
  end
end
```


heapify(A, 1, 5)
s=6
1 4 6 3 2 7
6 4 1 3 2 7
heapify(A, 1, 4)
s=5
2 4 1 3 6 7
4 3 1 2 6 7



heapify(A, 1, 3)
s=4
heapify(A, 1, 2)
s=3
s=2

# Complexity of heapsort

- Initial build heap: $O(n)$
- heapSort: $n$ iterations of heapify, each $O(\log n)$
- Total: $O(n) + O(n \log n) = O(n \log n)$
- And does the sorting in place!