Curtin
University of Technology

Design and Analysis of Algorithms

Lecture 4

Heaps

Curtin University of Technology

Topics

- Heaps
- Building Heaps
- Maintaining heaps
- Heapsort
  - Algorithm
  - Analysis
- Priority Queues

Heaps - Introduction

- The heap is a data structure for implementing a Partially Ordered Tree or Leftist tree (plus priority queue!)
  - Popular for use as an efficient priority queue
  - A heap is an example of a weakly ordered tree
  - It is special because its structure can be represented in an array
- Each node of a heap tree corresponds to an element of the array that stores the value in the node
  - The tree is stored implicitly
    - But it can be represented explicitly as a standard binary tree with left and right pointers, i.e., represented conceptually as a binary tree
    - The tree is filled on all levels except possibly the lowest, which are filled from left to right up to a point
- An array *A* that represents a heap is an object with two attributes:
  - *A.length* → the number of *allocated* elements in the array
  - *A.heap_size* → the number of elements in the heap stored within the array
  - $A.heap\_size \leq A.length$

Heaps (cont.)

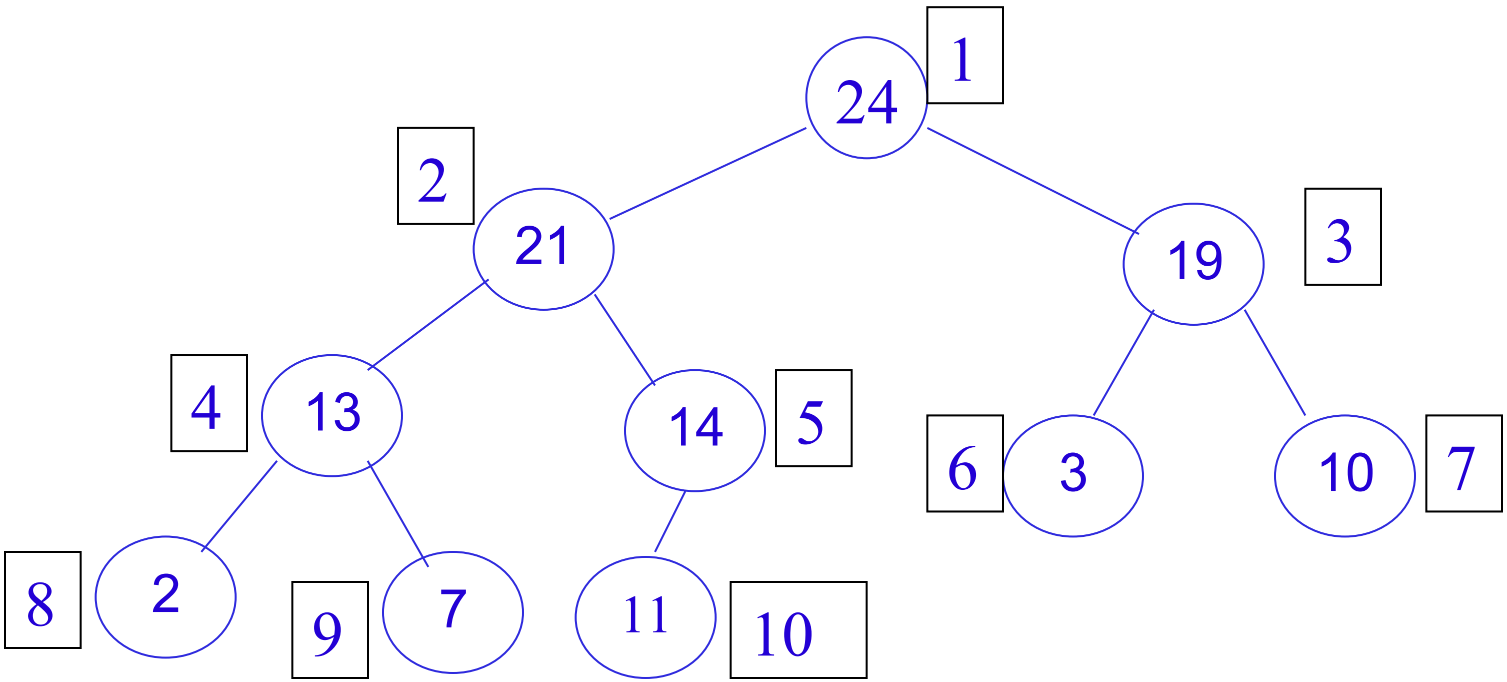
- $A[1]$ is the **root** of the tree
- For a node with index i:
  - PARENT(i) is the index of the parent of i
  - PARENT(i) = $\lfloor i/2 \rfloor$
  - LEFT_CHILD(i) is the index of the left child of i
  - LEFT_CHILD(i) = $2 \times i$
  - RIGHT_CHILD(i) is the index of the right child of i
  - RIGHT_CHILD(i) = $2 \times i + 1$

Example:

Index	1	2	3	4	5	6	7	8	9	10
Value	24	21	19	13	14	3	10	2	7	11

Curtin University of Technology

Heaps (cont.)



1	2	3	4	5	6	7	8	9	10
24	21	19	13	14	3	10	2	7	11

Binary Heap Properties

- Two kinds of binary heaps:

  Max-heap: $A[\text{PARENT}(i)] \geq A[i]$

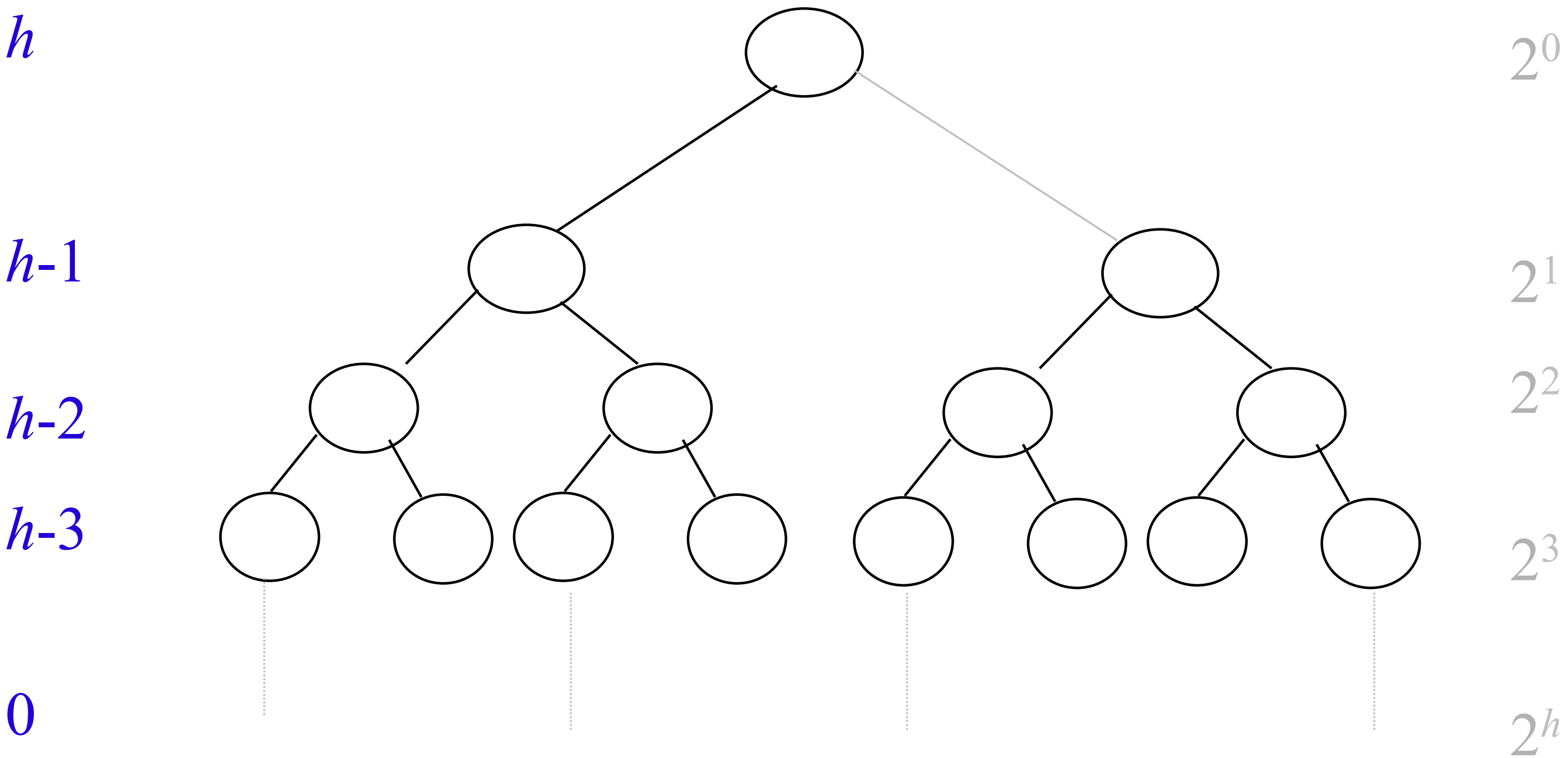
  Min-heap: $A[\text{PARENT}(i)] \leq A[i]$

- The binary heap is based on a binary tree
- The height of a heap is the number of edges on the longest simple downward path from the root to a leaf
- The height of a heap with n nodes is $O(\log n)$
- All basic operations on heaps run in $O(\log n)$ time

Curtin University of Technology

Binary Heap Tree

Height



$n=2^0+2^1+2^2+2^3+\ldots+2^h=2^{h+1}-1$ →Geometric Series

$$\sum_{i=0}^{h} x^i = \frac{x^{h+1}-1}{x-1} \Rightarrow \frac{2^{h+1}-1}{2-1} = 2^{h+1}-1$$

Heap Algorithms

MAX-HEAPIFY

- maintains the max-heap property
- $O(\log n)$

BUILD-MAX-HEAP

- produces a max-heap from an unordered input array
- $O(n)$

HEAPSORT

- sorts an array in place
- $O(n \log n)$

HEAP-EXTRACT-MAX

- used for priority queue
- $O(\log n)$

MAX-HEAP-INSERT

- inserts an element to the heap
- $O(\log n)$

MAX-HEAPIFY

- MAX-HEAPIFY checks the heap elements for violation of the heap property and restores the property.

MAX-HEAPIFY (A, i)

Input:

- An array A and index i to the array
  - $i = 1$ if we want to max-heapify the whole tree
- Sub-trees rooted at LEFT_CHILD(i) and RIGHT_CHILD(i) are heaps
  - Assume that the sub-trees are already Max-Heaps.

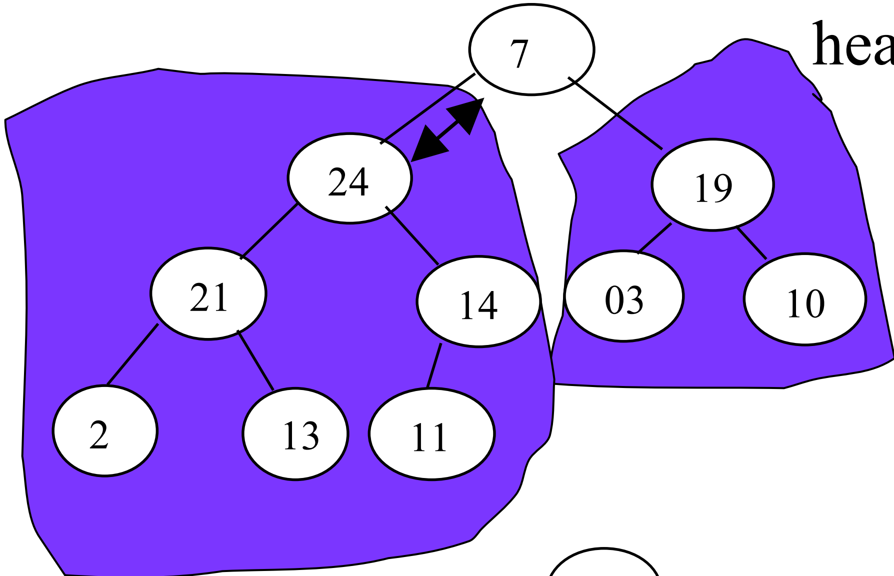
Output:

- The elements of array A forming a sub-tree rooted at i that satisfy the heap property.

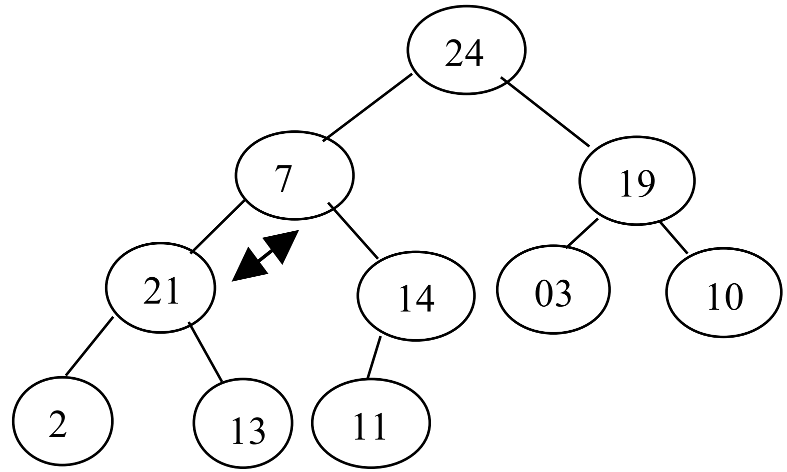
Curtin University of Technology

MAX-HEAPIFY (Trickle-down)

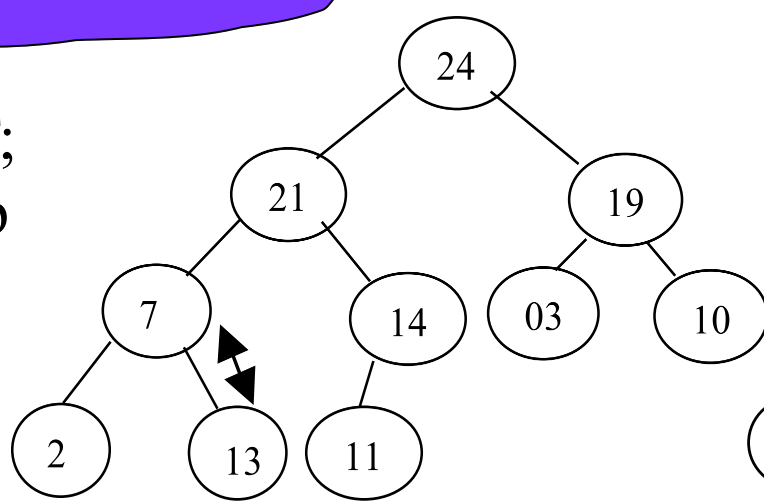
```
MAX-HEAPIFY (A, i)                        // trickle down process
1.  l = LEFT_CHILD (i)                    // left heap in relation to node i
2.  r = RIGHT_CHILD (i)                   // right heap in relation to node i
3.  if l ≤ A.heap_size and A[l] > A[i]
4.      then largest = l                  // parent > child
5.      else largest = i                  // parent <= child
6.  if r ≤ A.heap_size and A[r] > A[largest]
7.      then largest = r                  // lines 3-7 = choose largest of A[i], A[l], A[r]
8.  if largest ≠ i
9.     then exchange A[i] with A[largest]
10.     MAX-HEAPIFY (A, largest)          // continue trickle-down
```

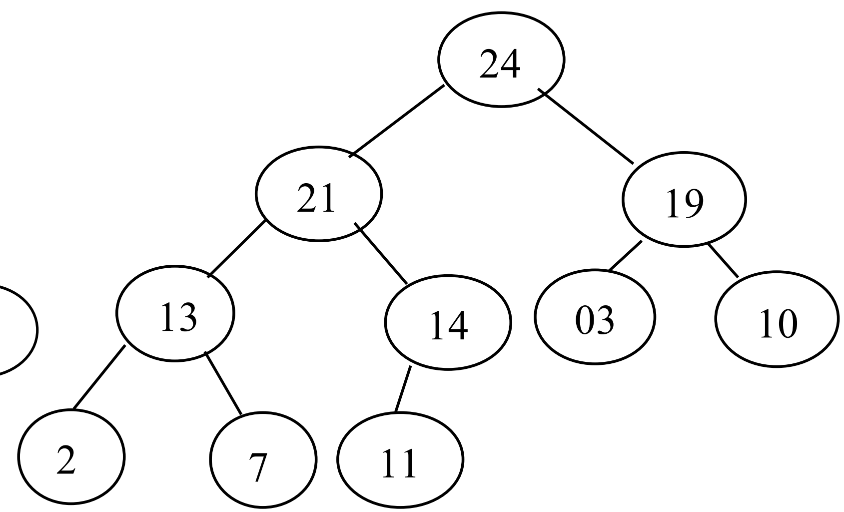


RST,
heap



LST;
heap





Running time of MAX-HEAPIFY

7	**24**	**19**	**21**	**14**	**03**	**10**	**02**	**13**	**11**
24	**7**	**19**	**21**	**14**	**03**	**10**	**02**	**13**	**11**
24	**21**	**19**	**07**	**14**	**03**	**10**	**02**	**13**	**11**
24	**21**	**19**	**13**	**14**	**03**	**10**	**02**	**07**	**11**

Total running time = steps 1 … 9 + recursive call

$T(n) = T(n/2) + \Theta(1)$ // recursion + steps

Solving the recurrence, we get $T(n) = O(\log n)$

→ Can use the Master Theorem

Running time of MAX-HEAPIFY

Show $T(n) = \Theta(1) + T\left(\lfloor n/2 \rfloor\right)$ is $O(\log n)$

Assume $T(x) \le c \log x$ holds when $x = \lfloor n/2 \rfloor$

$$
\begin{aligned}
T(n) \quad &\le \Theta(1) + c \log \lfloor n/2 \rfloor \\
&\le c \ \log n \quad \text{ if } c > 0
\end{aligned}
$$

Now for the base case when $n = 2$

$T(2) = \Theta(1) + T\left(1\right) = \Theta(1)$

Exercise: so what is n_0 and c?

Curtin University of Technology

BUILD-MAX-HEAP (*A*)

Input: An array *A* of size $n = A.length$; $A.heap\_size$

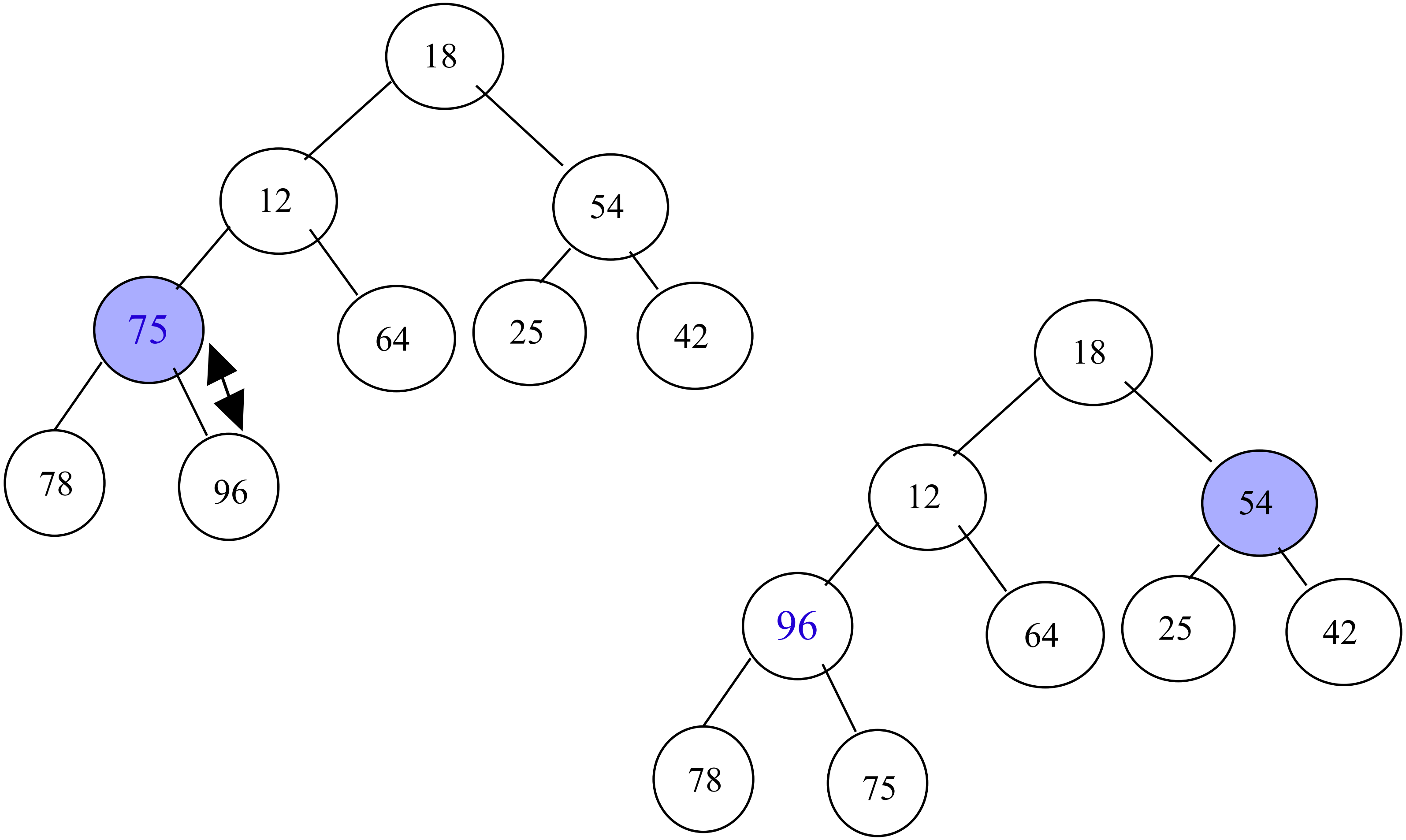
Output: A max-heap of size *n*

1. $A.heap\_size = A.length$
2. **for** $i = \lfloor A.length/2 \rfloor$ downto 1   // start at last non-leaf, go backwards
3. **do** MAX-HEAPIFY(*A*, *i*)   // put ith element in correct place in heap

18	12	54	75	64	25	42	78	96
18	12	54	96	64	25	42	78	75
18	12	54	96	64	25	42	78	75
18	96	54	12	64	25	42	78	75
18	96	54	78	64	25	42	12	75
96	18	54	78	64	25	42	12	75
96	78	54	18	64	25	42	12	75
96	78	54	75	64	25	42	12	18

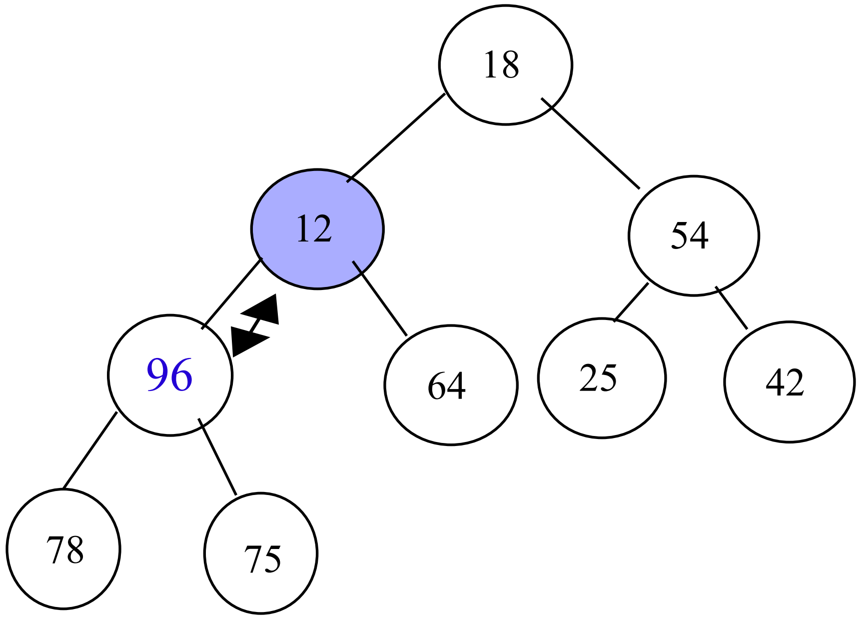
Curtin University of Technology

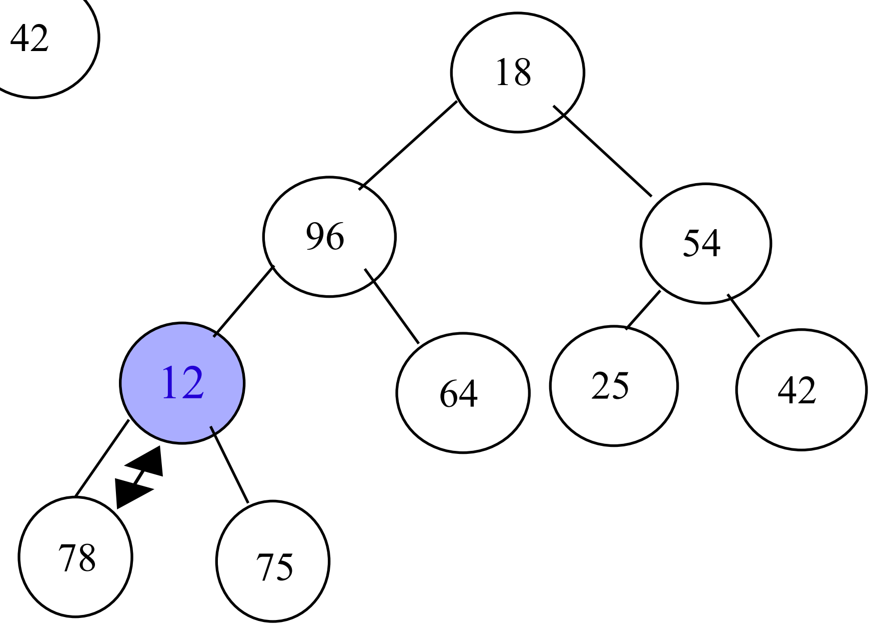
18 12 54 75 64 25 42 78 96



18 12 54 96 64 25 42 78 75

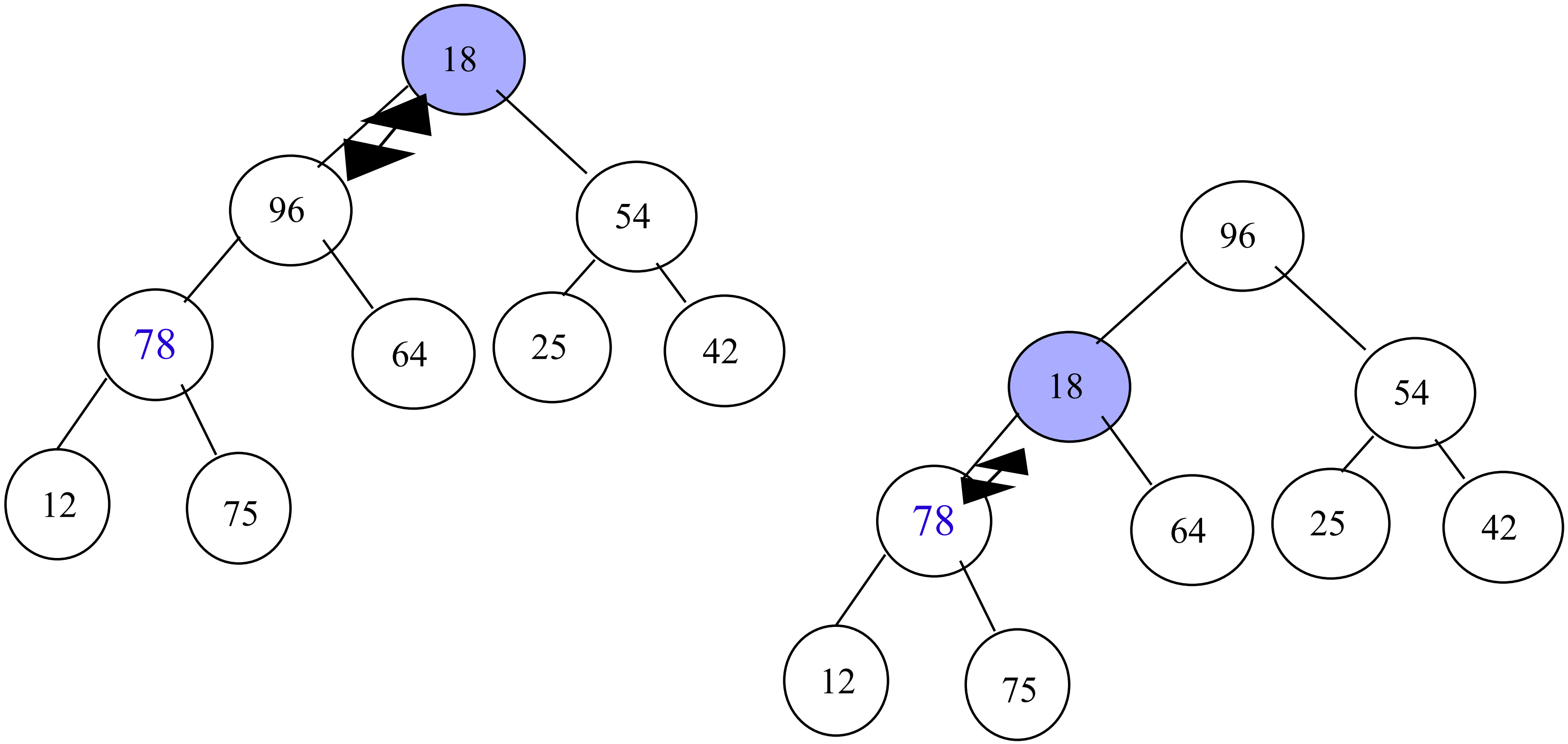
18 12 54 96 64 25 42 78 75





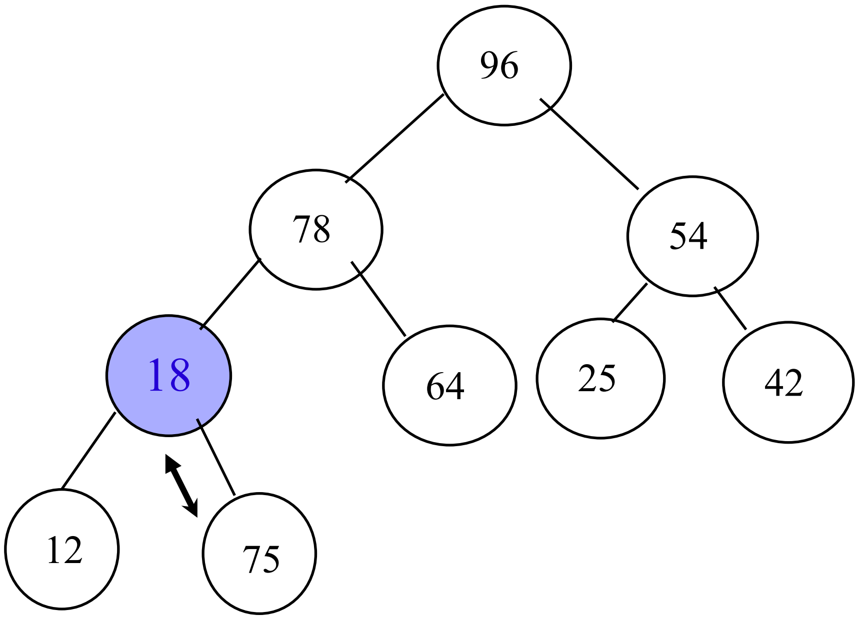
18 96 54 12 64 25 42 78 75

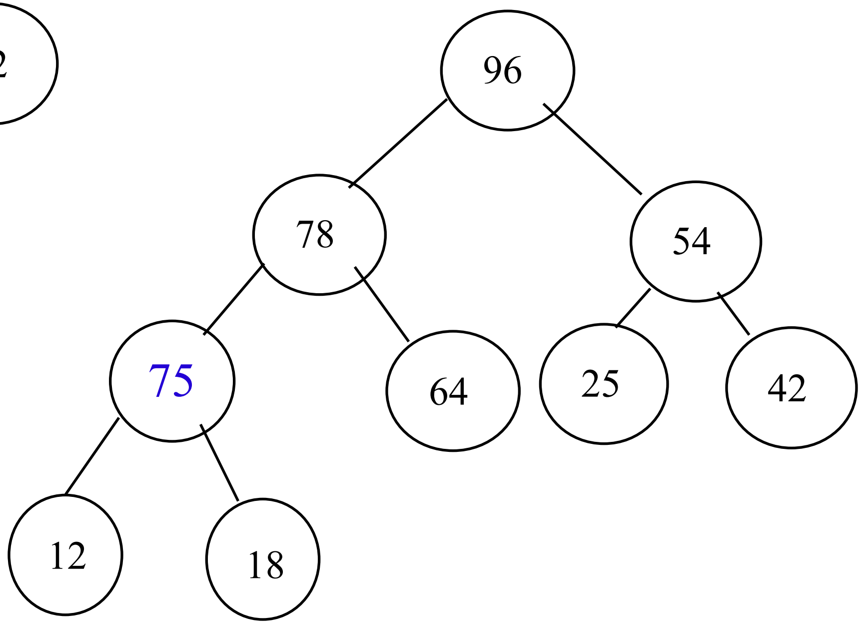
18 96 54 78 64 25 42 12 75

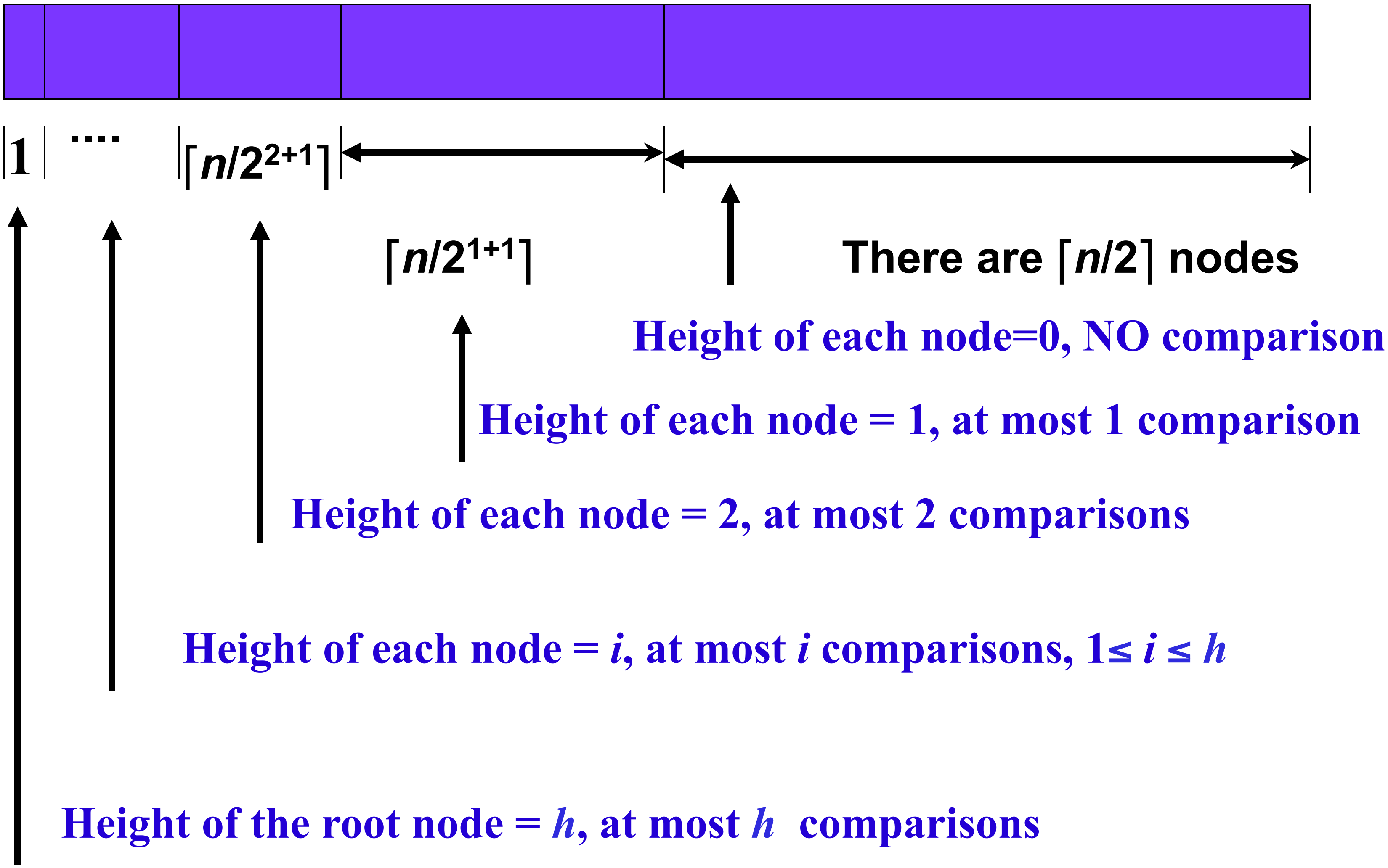


96 18 54 78 64 25 42 12 75

96 **78** **54** **18** **64** **25** **42** **12** **75**






1
....
$\lceil n/2^{2+1} \rceil$
$\lceil n/2^{1+1} \rceil$
There are $\lceil n/2 \rceil$ nodes
Height of each node=0, NO comparison
Height of each node = 1, at most 1 comparison
Height of each node = 2, at most 2 comparisons
Height of each node = i, at most i comparisons, $1 \leq i \leq h$
Height of the root node = h, at most h comparisons

Running time (Proof)

- In an n element heap there are at most $\lceil n/2^{h+1} \rceil$ nodes of height h
- The time required to heapify a sub-tree whose root is at a height h is $O(h)$ (this was proved in the analysis for MAX_HEAPIFY)
- So the total time taken for BUILD_MAX_HEAP is given by,

$$\sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot h \le \frac{n}{2} \cdot \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h} \le \frac{n}{2} \cdot \sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{n}{2} \cdot 2 = O(n)$$

Note: $$\sum_{h=0}^{\infty} \frac{h}{2^h} = 2$$

Thus the running time of BUILD_MAX_HEAP is given by O(n)

Curtin University of Technology

HEAPSORT

Idea:

- Build max-heap on $A[1 \,..n]$, where $n = A.length$
- Exchange $A[n]$ with $A[1]$; Note: $A[1]$ contains **max** element
- Discard $A[n]$ from the heap → decrement $A.heap\_size$
- Make $A[1 \,..\, n\text{-}1]$ into a max-heap; call MAX-HEAPIFY (A,1)
- Repeat the process down to a heap of size 2

HEAPSORT Algorithm

HEAPSORT(A)

Input: Array $A[1\ldots n]$, $n = A.length$

Output: Sorted array $A[1\ldots n]$

```
BUILD-MAX-HEAP[A]
for i = A.length downto 2
    do exchange A[1] with A[i]
        A.heap_size = A.heap_size - 1
        MAX-HEAPIFY(A,1)
```

HEAPSORT-example

Input

1	2	3	4	5	6	7	8	9
12	16	3	11	10	1	2	5	4

After Build

1	2	3	4	5	6	7	8	9
16	12	3	11	10	1	2	5	4

After 1st swap

1	2	3	4	5	6	7	8	9
4	12	3	11	10	1	2	5	**16**

After heapify

1	2	3	4	5	6	7	8	9
12	11	3	5	10	1	2	4	**16**

Curtin University of Technology

	1	2	3	4	5	6	7	8	9
After heapify	12	11	3	5	10	1	2	4	**16**

	1	2	3	4	5	6	7	8	9
After 2nd swap	4	11	3	5	10	1	2	**12**	**16**

	1	2	3	4	5	6	7	8	9
After heapify	11	10	3	5	4	1	2	**12**	**16**

	1	2	3	4	5	6	7	8	9
After 3rd swap	2	10	3	5	4	1	**11**	**12**	**16**

	1	2	3	4	5	6	7	8	9
After heapify	10	5	3	2	4	1	**11**	**12**	**16**

	1	2	3	4	5	6	7	8	9
After heapify	10	5	3	2	4	1	**11**	**12**	**16**

	1	2	3	4	5	6	7	8	9
After 4th swap	1	5	3	2	4	**10**	**11**	**12**	**16**

	1	2	3	4	5	6	7	8	9
After heapify	5	4	3	2	1	**10**	**11**	**12**	**16**

	1	2	3	4	5	6	7	8	9
After 5th swap	1	4	3	2	**5**	**10**	**11**	**12**	**16**

	1	2	3	4	5	6	7	8	9
After heapify	4	2	3	1	**5**	**10**	**11**	**12**	**16**

HEAPSORT-complexity

- **Running Time:**
  - Step 1: BUILD_MAX_HEAP takes $O(n)$ time
  - Steps 2 to 5: there are $(n$-1) calls to MAX_HEAPIFY which takes $O(\log n)$ time

Therefore running time takes $O(n \log n)$

Curtin University of Technology

Priority Queues

A priority queue is:

- An abstract data type which consists of a set of elements.
- Each element of the set has an associated priority or key
- Priority is the value of the element or value of some component of an element

Operations performed on priority queues:

- Inserting an element into the set → HEAP_INSERT
- Finding and deleting from the set an element of highest priority → HEAP_EXTRACT_MAX
- Merging priority queues

Curtin University of Technology

Example

S: {(Brown, 20), (Gray, 22), (Green, 21)} → priority based on name
{(Brown, 20), (Green,21), (Gray, 22)} → priority based on age

Each element could be a record and the priority could be based on one of the fields of the record

A Student's record:

Attributes :	**Name**	**Age**	**Sex**	**Student No.**	**Marks**
Values :	**John Brown**	**20**	**M**	**94XYZ23**	**75**

Priority can be based on name, age, student number, or marks

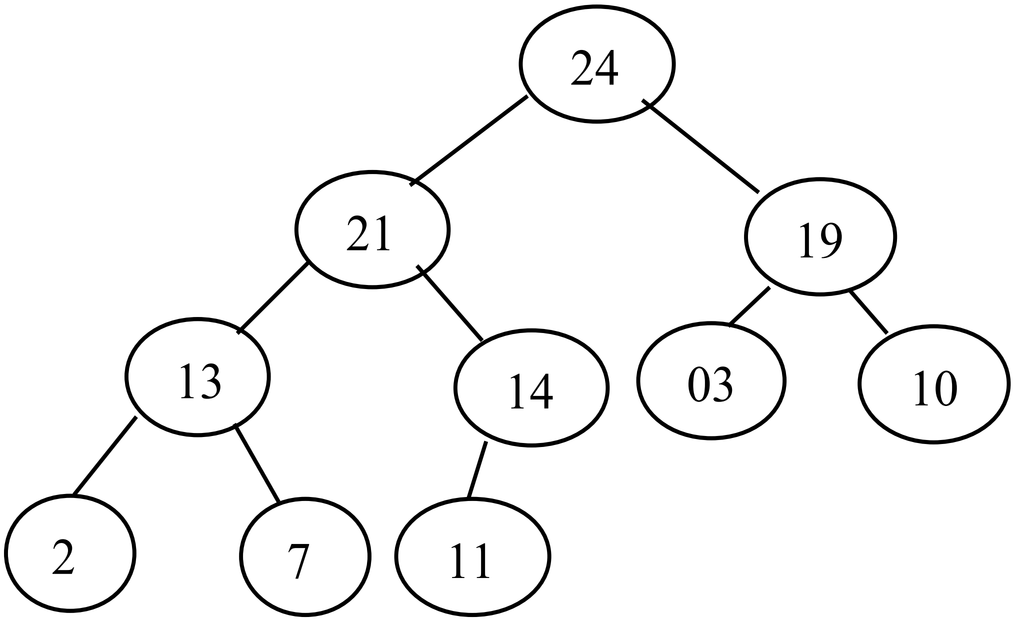
Curtin University of Technology

Priority Queues (cont.)

Priority queues are implemented on partially ordered trees (POTs).

- POTs are labeled binary trees.
- The labels of the nodes are elements with a priority.
- The element stored at a node has at least as large a priority as the elements stored at the children of that node.
- The element with the highest priority is at the root of the tree.

Example



HEAP_EXTRACT_MAX

Procedure HEAP_EXTRACT_MAX(*A*[1…*n*])
Input : heap(*A*)
Output : The maximum element or root, and heap (*A*[1…*n*-1])

1. **if** *A.heap_size* ≥ 1
2.   max = *A*[1];
3.   *A*[1] = *A*(*A.heap_size*);
4.   *A.heap_size* = *A.heap_size* - 1;
5.   MAX-HEAPIFY(*A*,1)
6.   **return** max

Running Time : $O(\lg n)$ time

Curtin University of Technology

HEAP_INSERT

HEAP_INSERT(*A*, *key*)
Input: heap ($A[1 \ldots n]$), *key* - the new element
Output: heap ($A[1 \ldots n+1]$) with *key* in the heap

1. $A.heap\_size = A.heap\_size + 1$;
2. $i = A.heap\_size$;
3. **while** $i > 1$ and $A[\text{PARENT}(i)] < key$
4. $\quad A[i] = A[\text{PARENT}(i)]$;
5. $\quad i = \text{PARENT}(i)$;
6. $\quad A[i] = key$

Example?
Running Time: $O(\lg n)$ time

Merging two PQs

- What is the cost of merging two PQs that are implemented as heaps
  - Assume m elements in one and n in the other
- One possible way:

BRUTE-FORCE-MERGE (A, *n*, B, *m*)

for $i = 1$ to n

HEAP_INSERT (B, HEAP_EXTRACT_MAX(A))

Analysis:

i	**Cost of Extract**	**Cost of insert**
1	$\log n$	$\log m$
2	$\log(n - 1)$	$\log (m+1)$
3	$\log(n - 2)$	$\log(m+2)$
…	…	…
n	$O(1)$	$\log(m+n)$

$$\sum_{i=1}^{n} \log i + \sum_{i=m}^{m+n} \log i$$

Curtin University of Technology

Merging two PQs – Analysis (cont.)

i	Cost of Extract	Cost of insert
1	$\log n$	$\log m$
2	$\log(n-1)$	$\log(m+1)$
3	$\log(n-2)$	$\log(m+2)$
…	…	…
n	$O(1)$	$\log(m+n)$

$$\sum_{i=1}^{n} \log i + \sum_{i=m}^{m+n} \log i$$

$$= \sum_{i=1}^{n} \log i + \sum_{i=1}^{m+n} \log i - \sum_{i=1}^{m} \log i$$

$$= O(n \log n) + O((m+n)\log(m+n)) - O(m \log m)$$

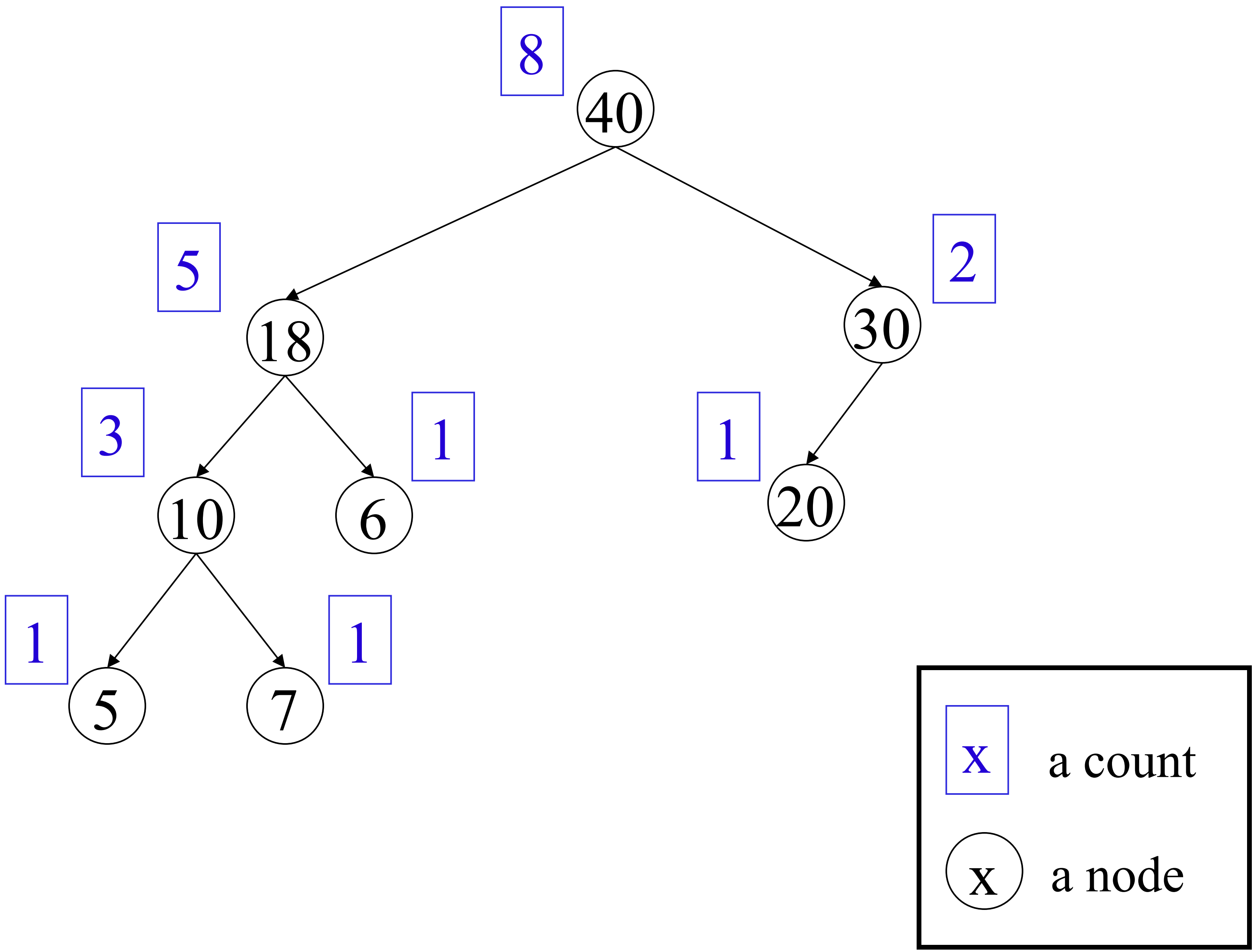
$$= O((m+n)\log(m+n))$$

Can we improve this?

Curtin University of Technology

Leftist tree

- Has heap condition:
  - all children less than parent (max heap)
  - all children greater than parent (min heap)
- *count* of nodes in left tree ≥ *count* in right
  - *count* is 1 + sum of child *counts*


8
40
5
18
2
30
3
1
1
10
6
20
1
1
5
7
x
a count
x
a node

How does that help?

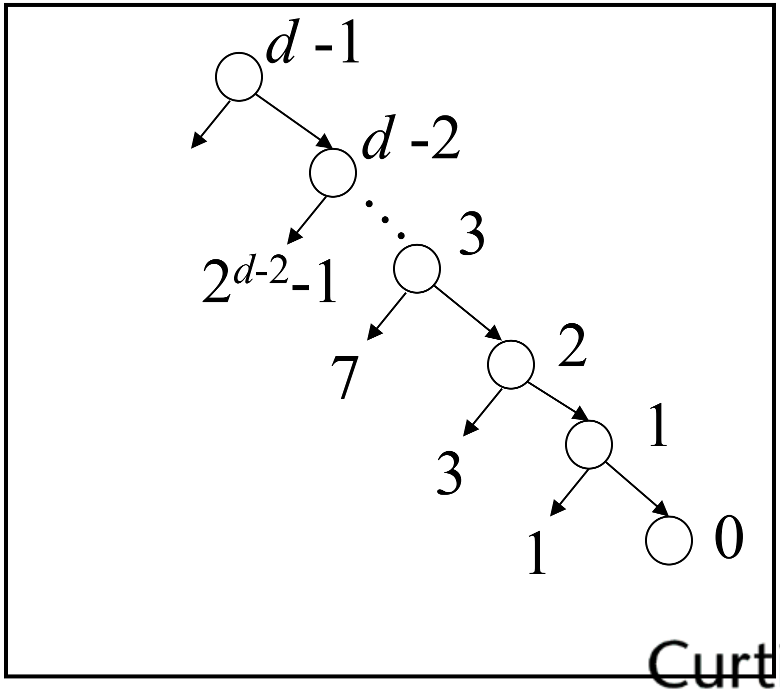
- Length of rightmost path from root is at most lg(n+1) → when the leftist tree is a complete binary tree with n nodes.
- **Proof by contradiction**
  - Assume there is a depth lg(n+1) + 1 right node
  - To keep the leftist tree property, we have to add in the min number of left nodes that exist
  - As a result there are more than n nodes in total, and therefore original assumption wrong

Curtin University of Technology

Alternative proof

- Assume there is a right path of length d
- Label nodes on the path d - 1 (the root) down to 0 (the rightmost node)
- Each left tree must have at least 2^i - 1 left nodes; for $i = 0$, 1, 2, … d - 1.

$$\sum_{i=0}^{d-1}(2^i - 1) \le n - d$$



- Use *Geometric Series* to show $d \le \lg(n + 1)$

So ...

- We can join two leftist trees by only traversing right path
  - Only do $O(1)$ work for each node
- If joining two trees of n and m items then visit at most lg $(n+1)$ + lg $(m+1)$ nodes
  - So we can join them in $O(\lg mn)$ time
  - If $n \geq m$, merge costing is $O(\lg n)$
- However, a leftist tree needs pointers while a binary heap does not need explicit pointers.

Curtin University of Technology

Note

$$\lg m \le c \lg (m+n) \quad \forall m > m_0 \text{ and } n > n_0$$

$$\lg n \le c' \lg (m+n) \quad \forall m > m_1 \text{ and } n > n_1 \qquad +$$

$$\lg mn = \lg m + \lg n \le (c + c')\lg(m+n)$$

$$\forall m > \max(m_0, m_1) \text{ and } n > \max(n_0, n_1)$$

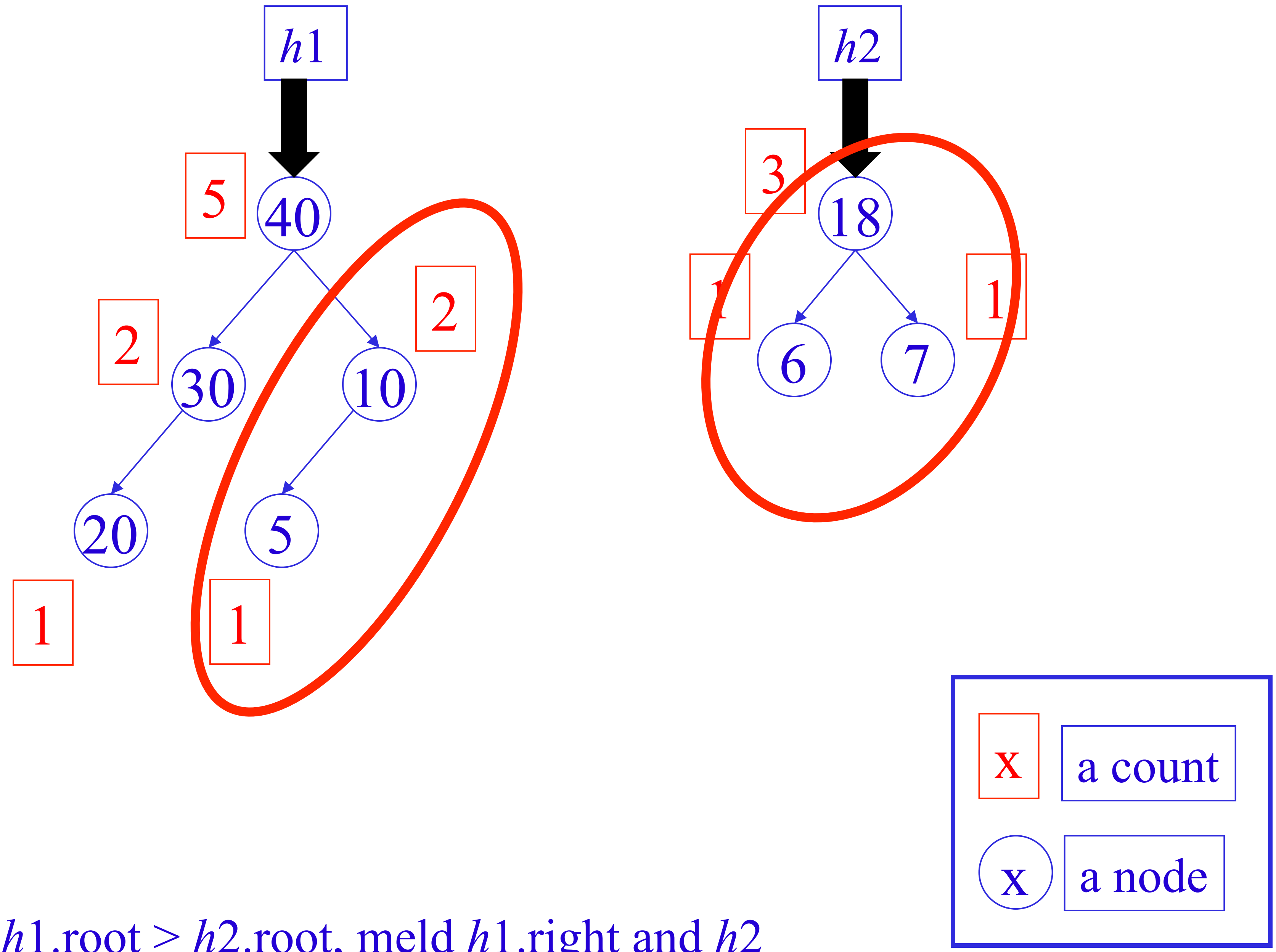
$$\Rightarrow O(\lg mn) \text{ is } O(\lg(m+n))$$

Curtin University of Technology

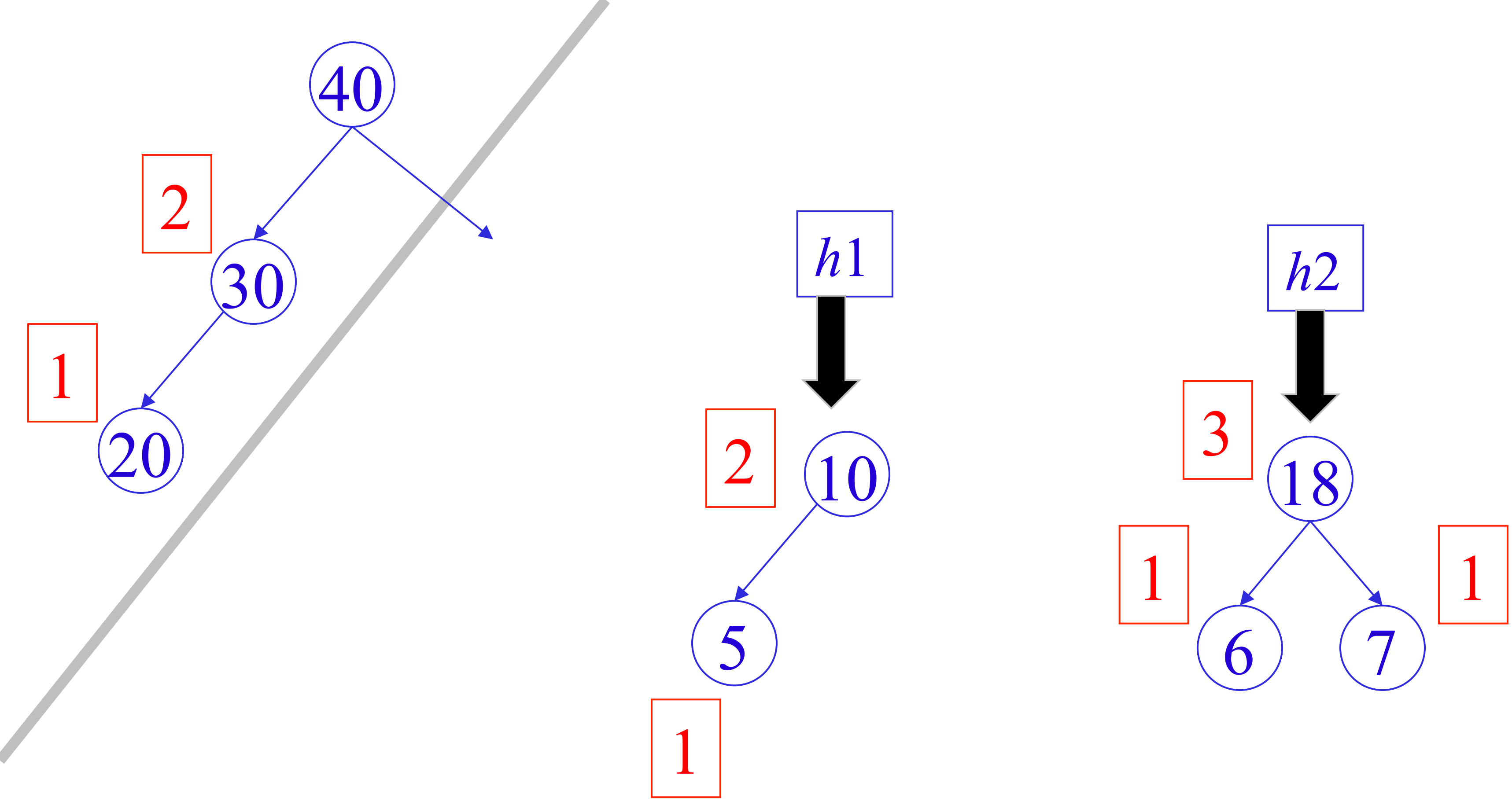
MELD Algorithm

```
MELD (h1, h2)
    if h1 is empty return h2
    if h2 is empty return h1
    if (h1.root < h2.root)
        swap(h1, h2)
    h1.right = MELD (h1.right, h2)
    if (h1.left.count < h1.right.count)
        swap(h1.left, h1.right)
    update h1.count
```

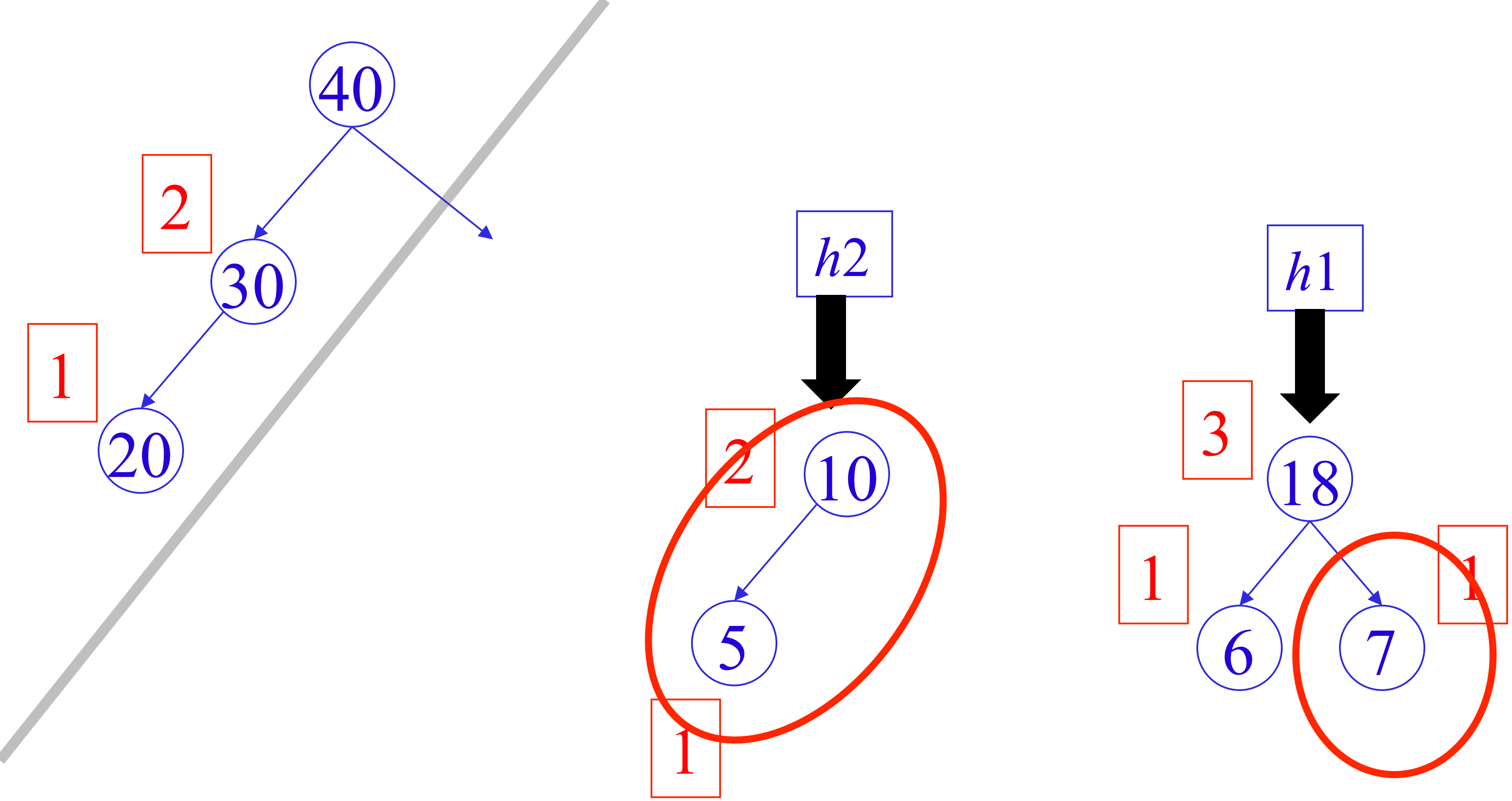
Curtin University of Technology



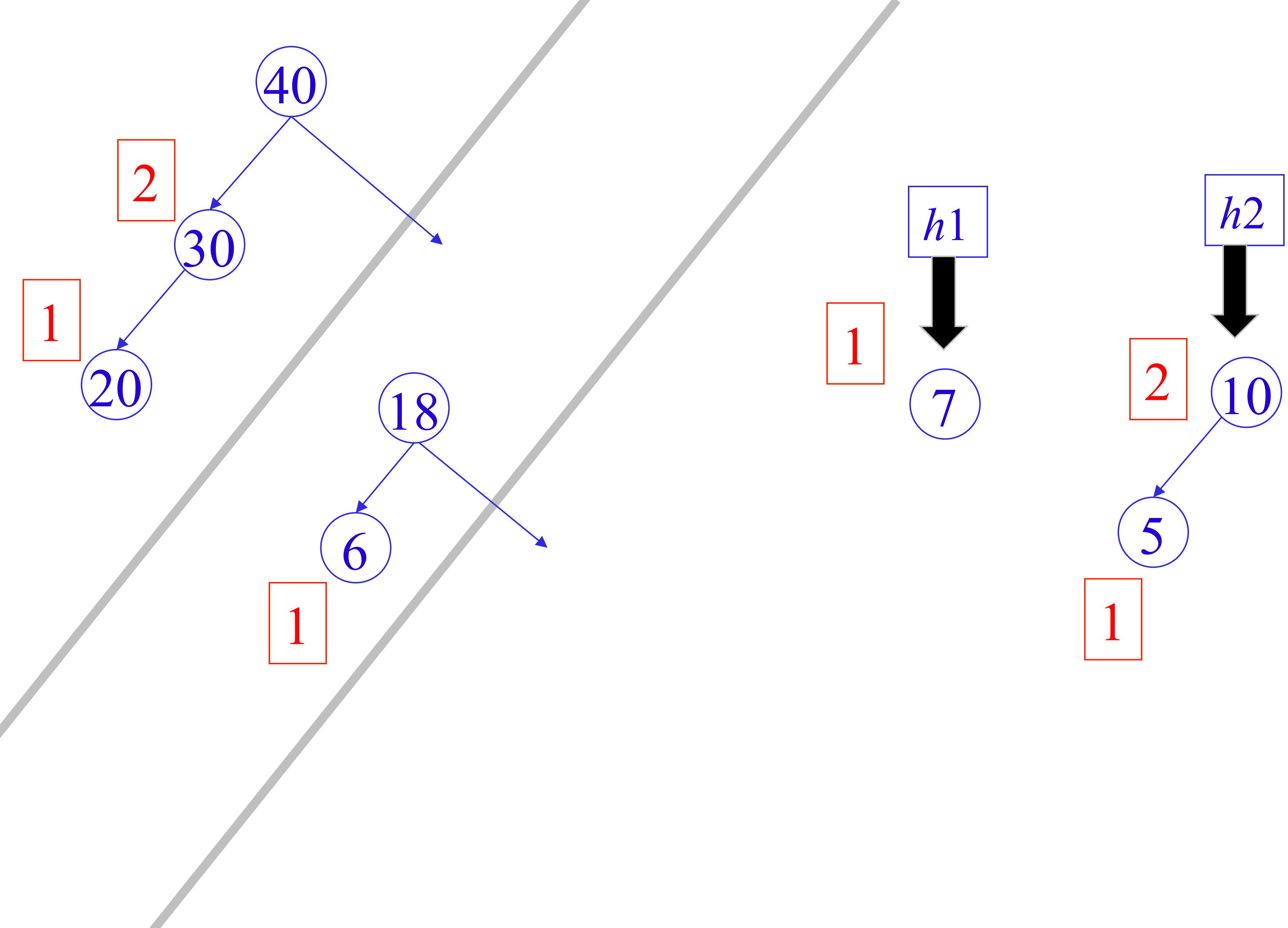
$h1$.root > $h2$.root, meld $h1$.right and $h2$



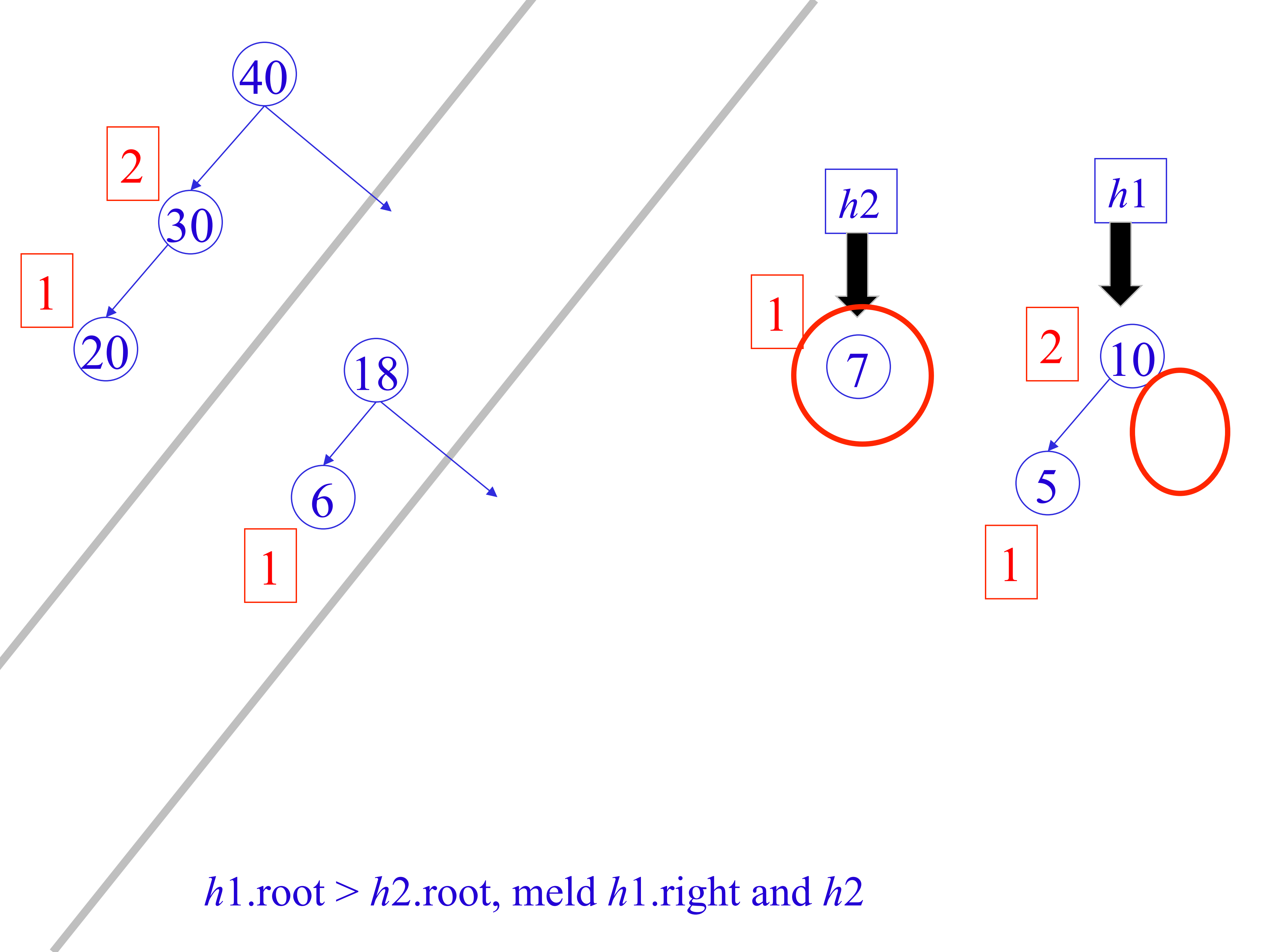
*h*1.root < *h*2.root, swap *h*1 and *h*2



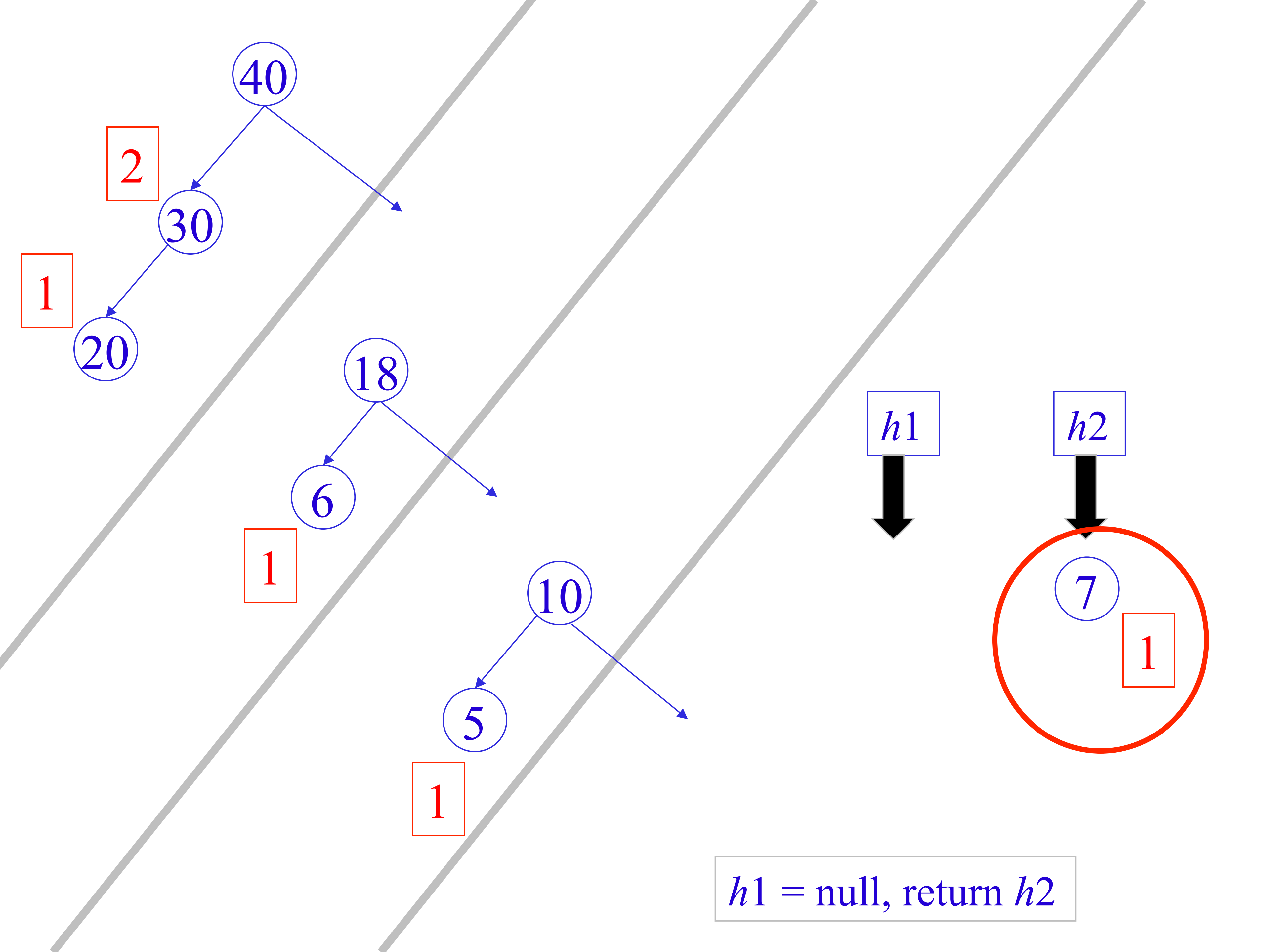
*h*1.root > *h*2.root, meld *h*1.right and *h*2

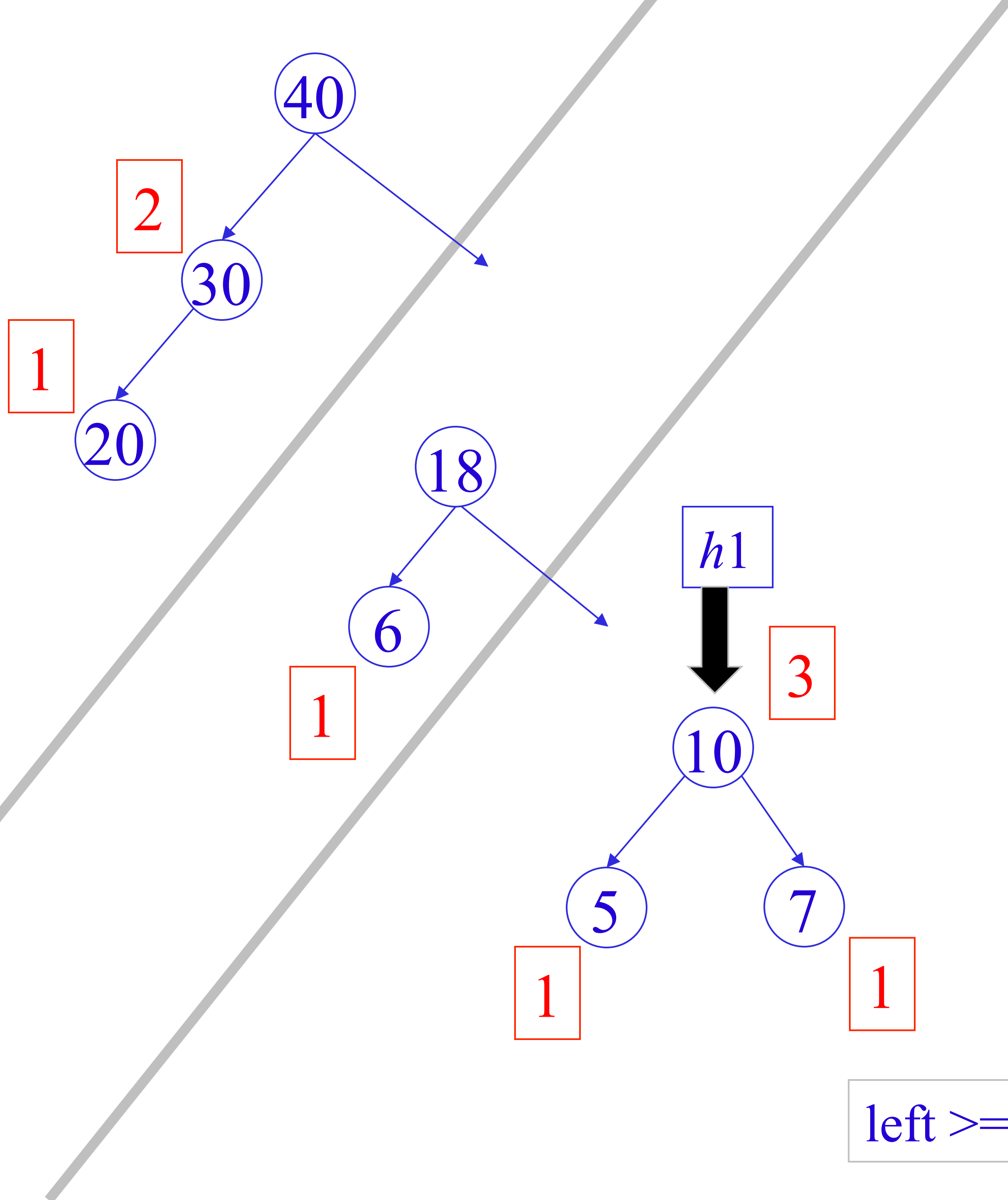


$h1$.root $<$ $h2$.root, swap $h1$ & $h2$

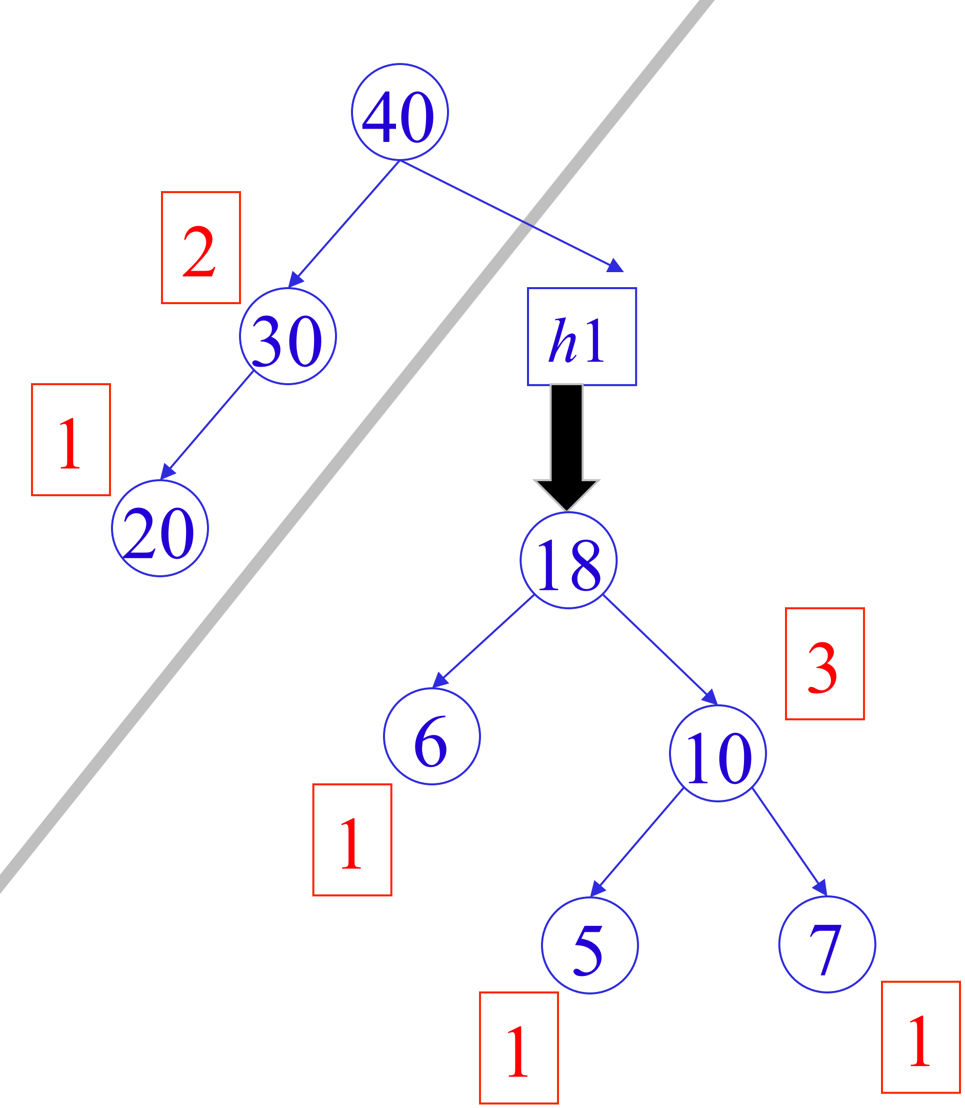


$h1$.root > $h2$.root, meld $h1$.right and $h2$

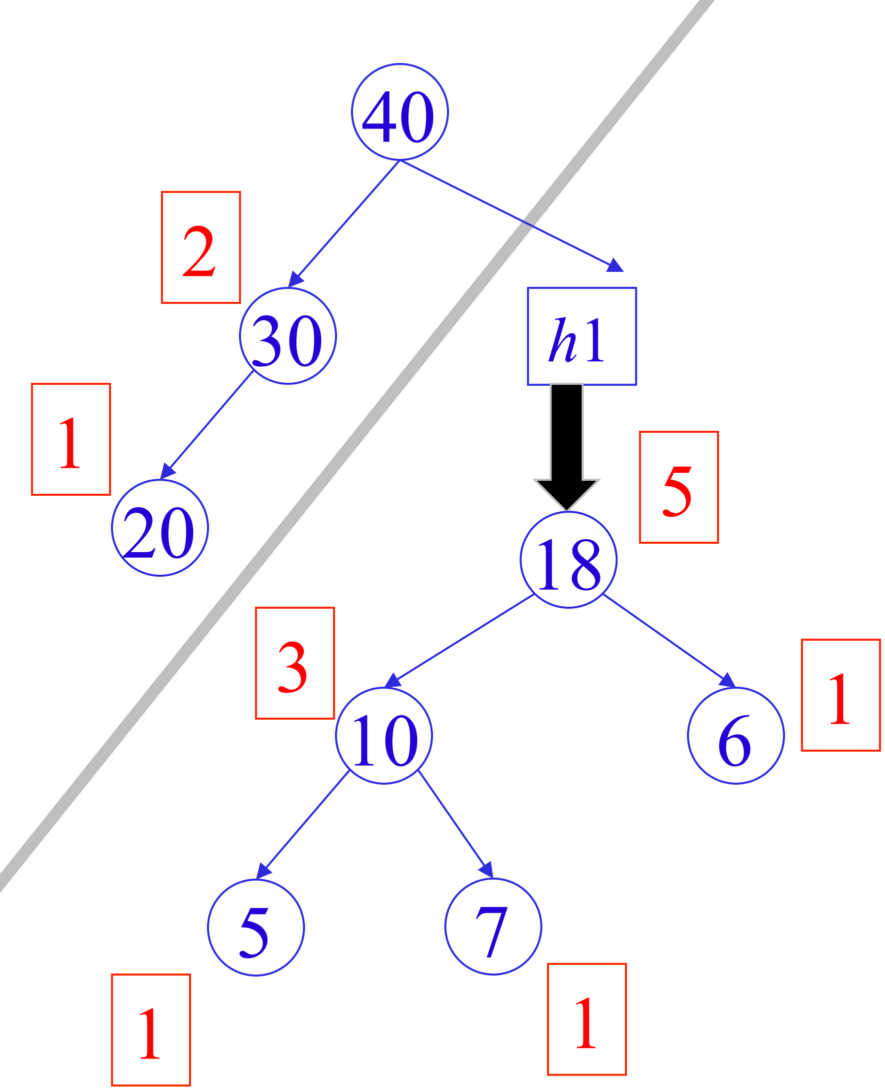

40
2
30
1
20
18
6
1
10
5
1
h1
h2
7
1
h1 = null, return h2



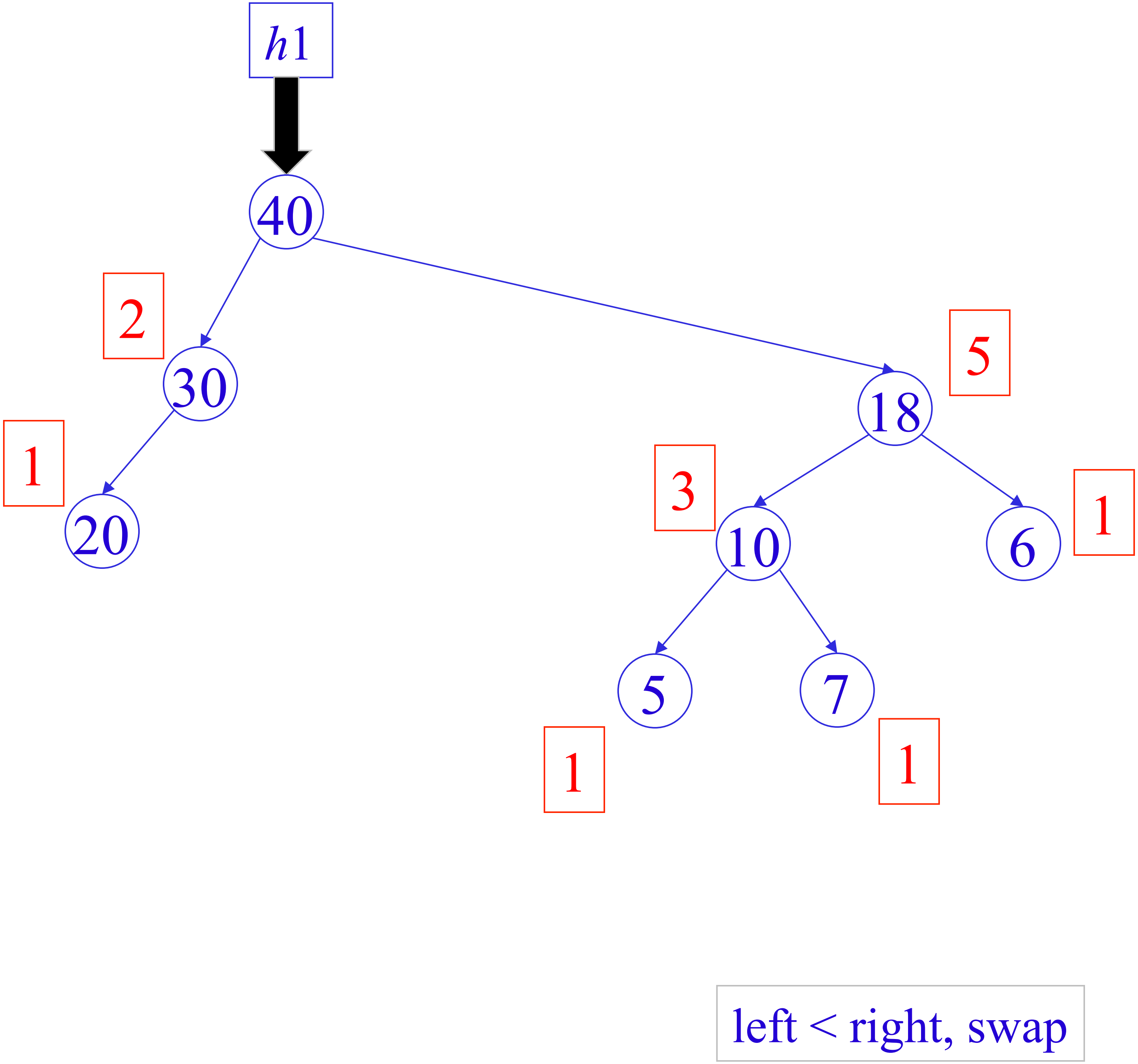
left >= right, update $h1$ count



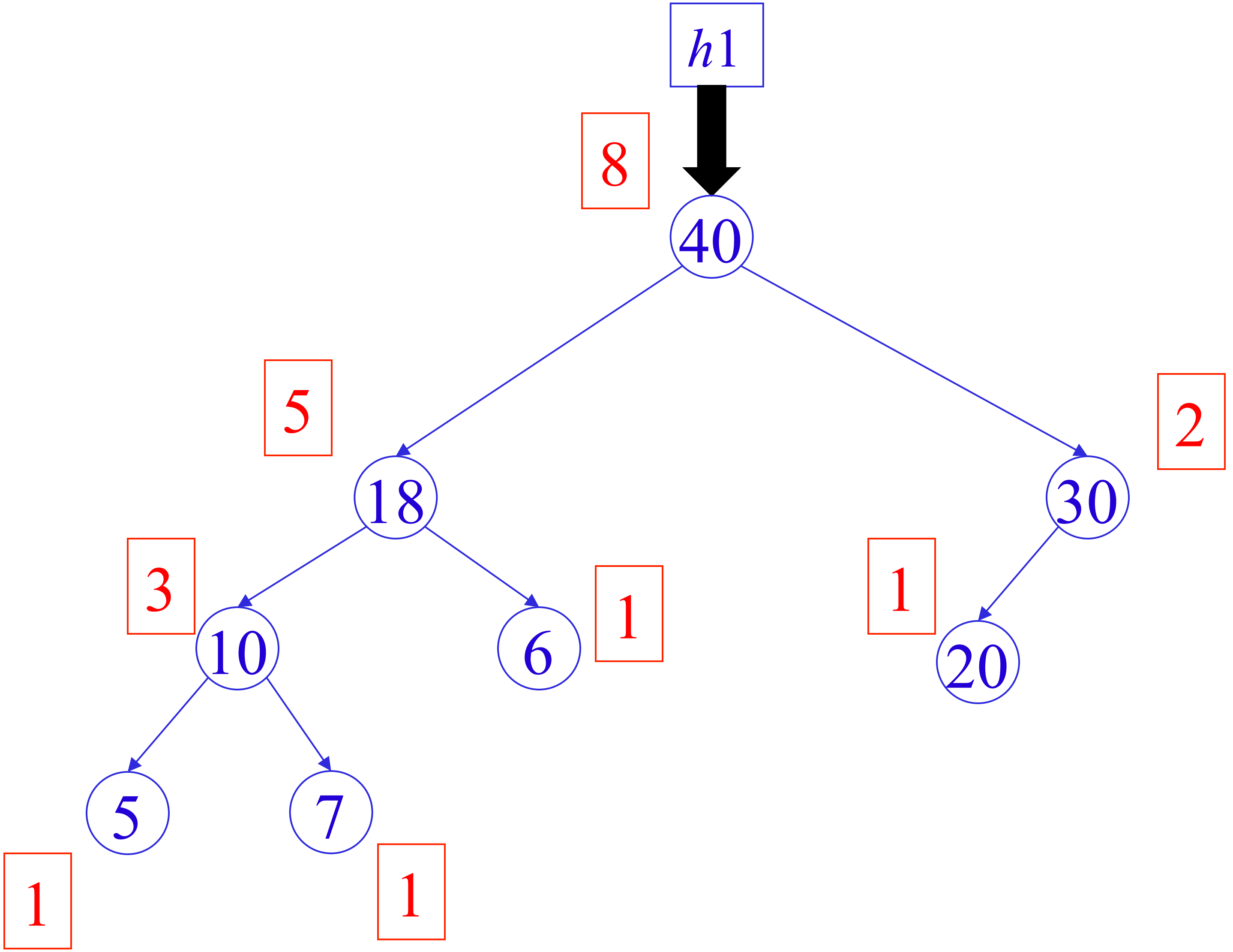
left < right, swap



left >= right, update $h1$ count



h1
40
30
2
20
1
18
5
10
3
6
1
5
1
7
1
left < right, swap



left >= right, update $h1$ count

MELD -complexity

MELD($h1$, $h2$)	$O(1)$
if $h1$ is empty **return** $h2$	$O(1)$
if $h2$ is empty **return** $h1$	$O(1)$
if ($h1$.root < $h2$.root)	$O(1)$
swap($h1$, $h2$)	$O(1)$
$h1$.right = **MELD**($h1$.right, $h2$)	$O(1)$
if ($h1$.left.count < $h1$.right.count)	$O(1)$
swap($h1$.left, $h1$.right)	$O(1)$
update $h1$.count	$O(1)$

Called log m + log n times = $O(\log mn) = O(\log(m+n))$

Curtin University of Technology

Other algorithms

- **INSERT(A, k)** = MELD(A, new node containing k)
- **EXTRACT-MAX(A)** = MELD(A.left, A.right)
- **BUILD-LEFT-TREE(B[1..n], A)**
  - MELD(B[1], B[2]) … MELD(B[n-1],B[n])
  - then MELD n/4 trees of 4 nodes
  - then MELD n/8 trees of 8 nodes
  - *etc.*