Heap

Heap Sort

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

ADT Heap

- A heap is a binary tree with
  - A structural property
    - Almost complete and almost balanced
      - All levels are complete, possibly except the last one, filled from left to right
  - A functional property
    - For each node different from the root we have that the key of the node is larger/less than the key of the parent node

ADT Heap

- Minimum heap

$$key[parent(node)] < key(node)$$

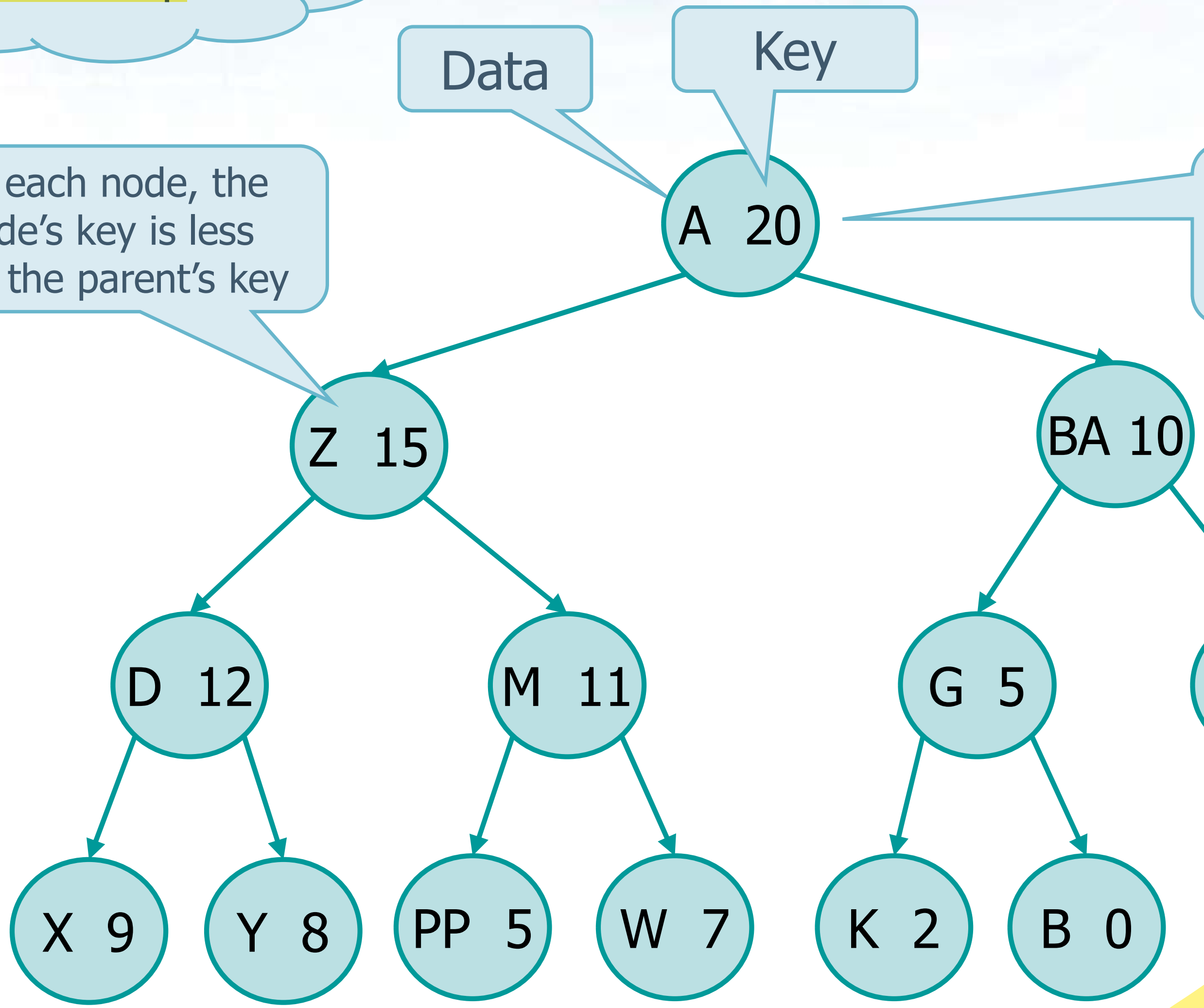
  - For mimimum heap the minimum key is in the root
- Maximum heap

$$key[parent(node)] > key(node)$$

  - For maximum heap the maximum key is in the root

Example

Max Heap

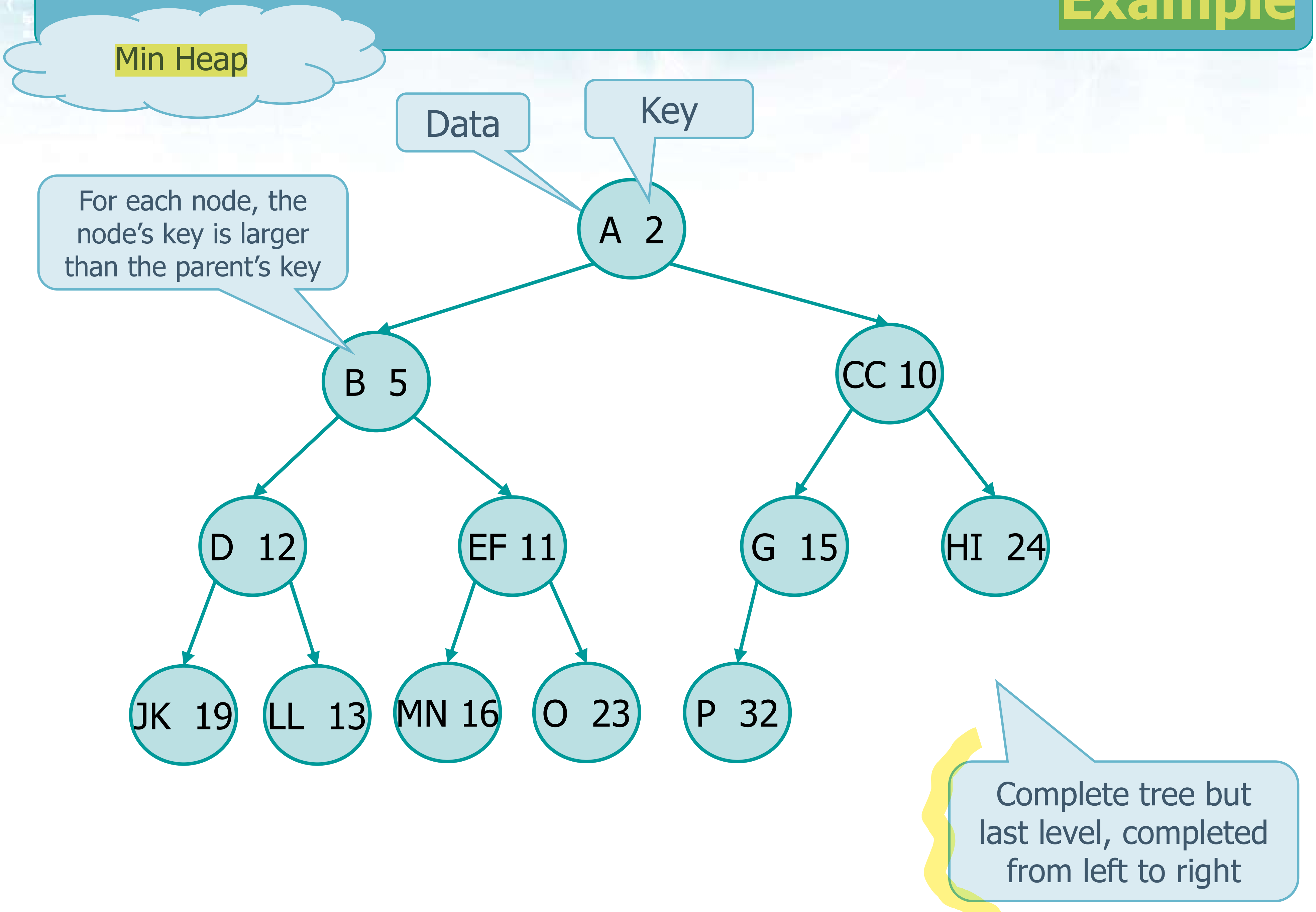


For each node, the node's key is less than the parent's key

We concentrate on **max heap** in this presentation

Complete tree but last level, completed from left to right

Example


Min Heap
Data
Key
For each node, the node's key is larger than the parent's key
A 2
B 5
CC 10
D 12
EF 11
G 15
HI 24
JK 19
LL 13
MN 16
O 23
P 32
Complete tree but last level, completed from left to right

ADT Heap

- A heap can be stored in an array of items
- The heap's wrapper can be defined as

```
struct heap_s {
  Item *A;
  int heapsize;
} heap_t;
```

The array A of maxN Items store the items (keys and data fields)

Heapsize specifiy the humber of elements stored in the heap heap->A

ADT Heap

- Given a node i, we define

```
#define LEFT(i)    (2*i+1)
#define RIGHT(i)   (2*i+2)
#define PARENT(i)  ((int)(i-1)/2)
```

```
#define LEFT(i)    (i<<1+1)
#define RIGHT(i)   (i<<1+2)
#define PARENT(i)  ((i-1)>>1)
```

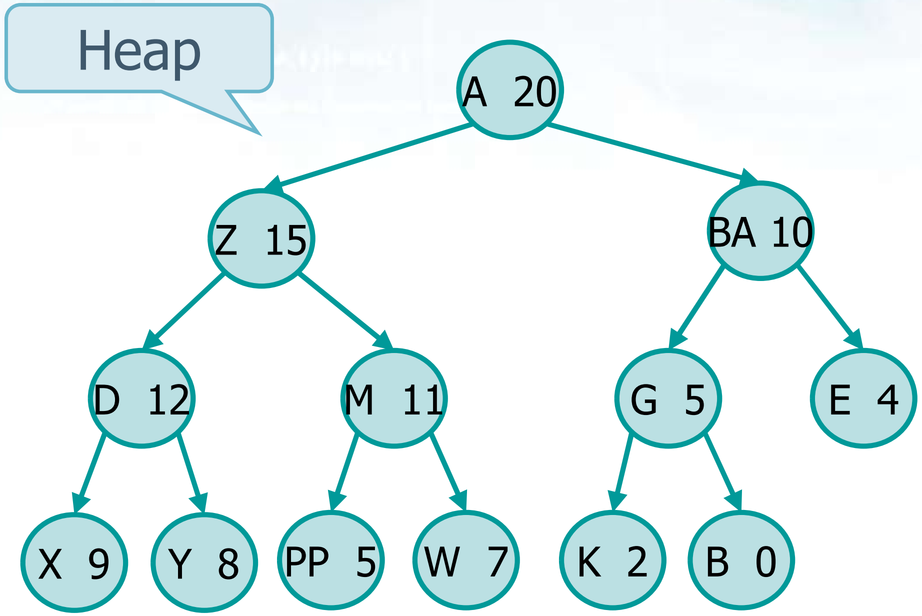
- Thus, given a node heap->A[i]

`heap->A[LEFT(i)]`	is its left child
`heap->A[RIGHT(i)]`	is its right child
`heap->A[PARENT(i)]`	is its parentd

  - The root of the heap is stored in

```
heap->A[0]
```

Example




```
#define LEFT(i)    (2*i+1)
#define RIGHT(i)   (2*i+2)
#define PARENT(i)  ((int)(i-1)/2)
```

```
struct heap_s {
  Item *A;
  int heapsize;
} heap_t;
```

Array representation

heap->A

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	Z	BA	D	M	G	E	X	Y	PP	W	K	B		
20	15	10	12	11	5	4	9	8	5	7	2	0		

heap->heapsize = 13

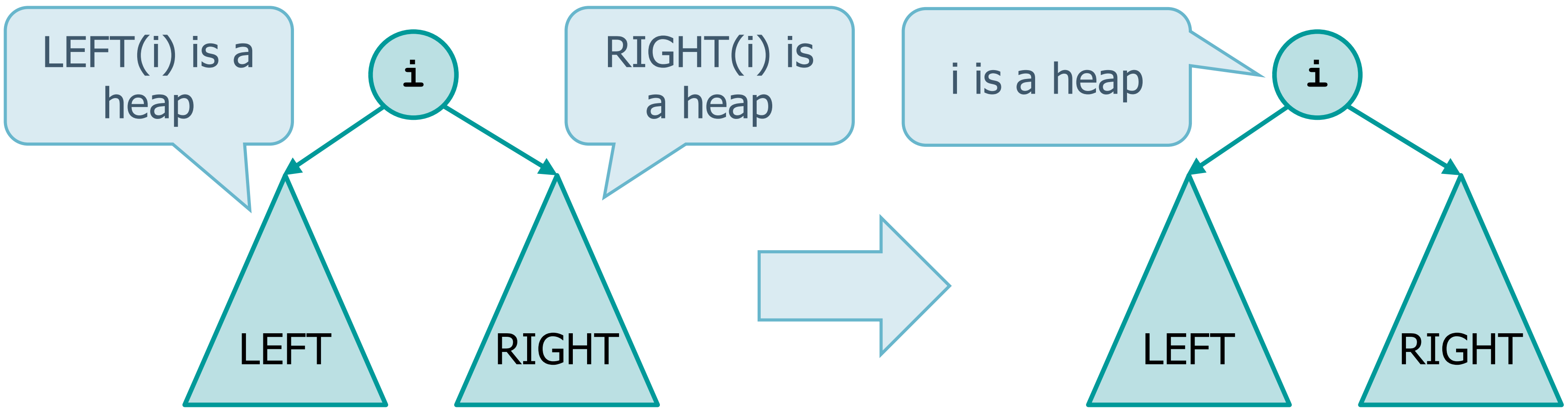
Array (maximum) maxN = 15

Heap sort

- Proposed in 1964 by the Welsh-Canadian computer scientist John William Joseph Williams (1930-2012)
- It is implemented through 3 main functions
  - Heapify
  - Heap-build
  - Heap-sort
- These functions call each other to elegantly build-up the final ordering on the same initial array

Function heapify

- **Premises**
  - Given a given node i, its sub-trees LEFT(i) and RIGHT(i) are already heaps
- **Outcome**
  - Turn into a heap the entire tree rooted at i, i.e., node i, with sub-trees LEFT(i) and RIGHT(i)



Function heapify

- Process
  - Compare A[i], LEFT(i) and RIGHT(i)
    - Assign to A[i] the maximum among A[i], LEFT(i) and RIGHT(i)
  - If there has been a swap between A[i] and LEFT(i)
    - Recursively apply heapify on the subtree whose root is LEFT(i)
  - If there has been a swap between A[i] and RIGHT(i)
    - Recursively apply heapify on the subtree whose root is RIGHT(i)
- Complexity
  - $T(n) = O(log_2 n)$

Height of the node ($log_2 n$) for the entire tree

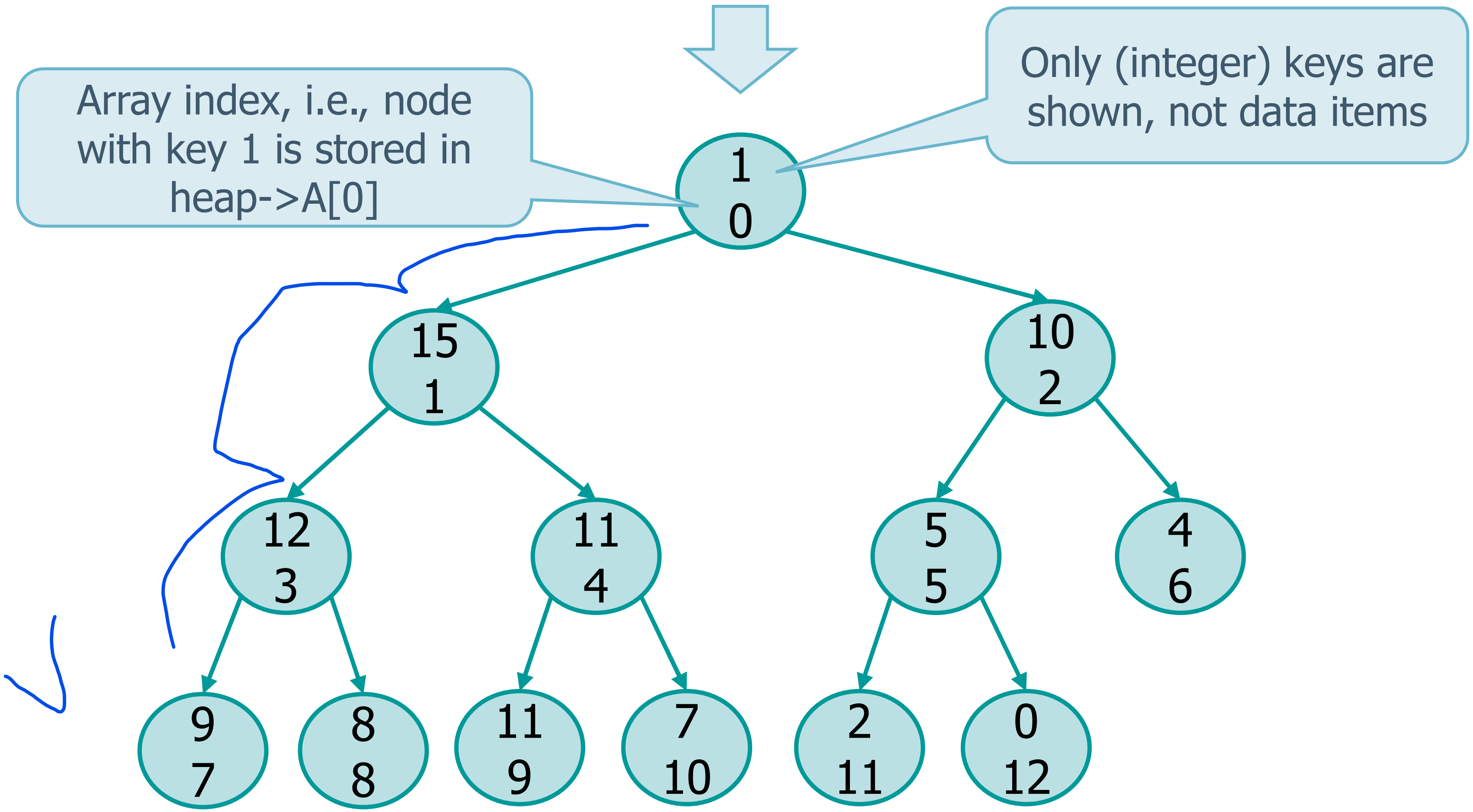
Example

- Given the following heap, show the result of
  - heapify (A, 0)

	0	1	2	3	4	5	6	7	8	9	10	11	12
A	1	15	10	12	11	5	4	9	8	11	7	2	0

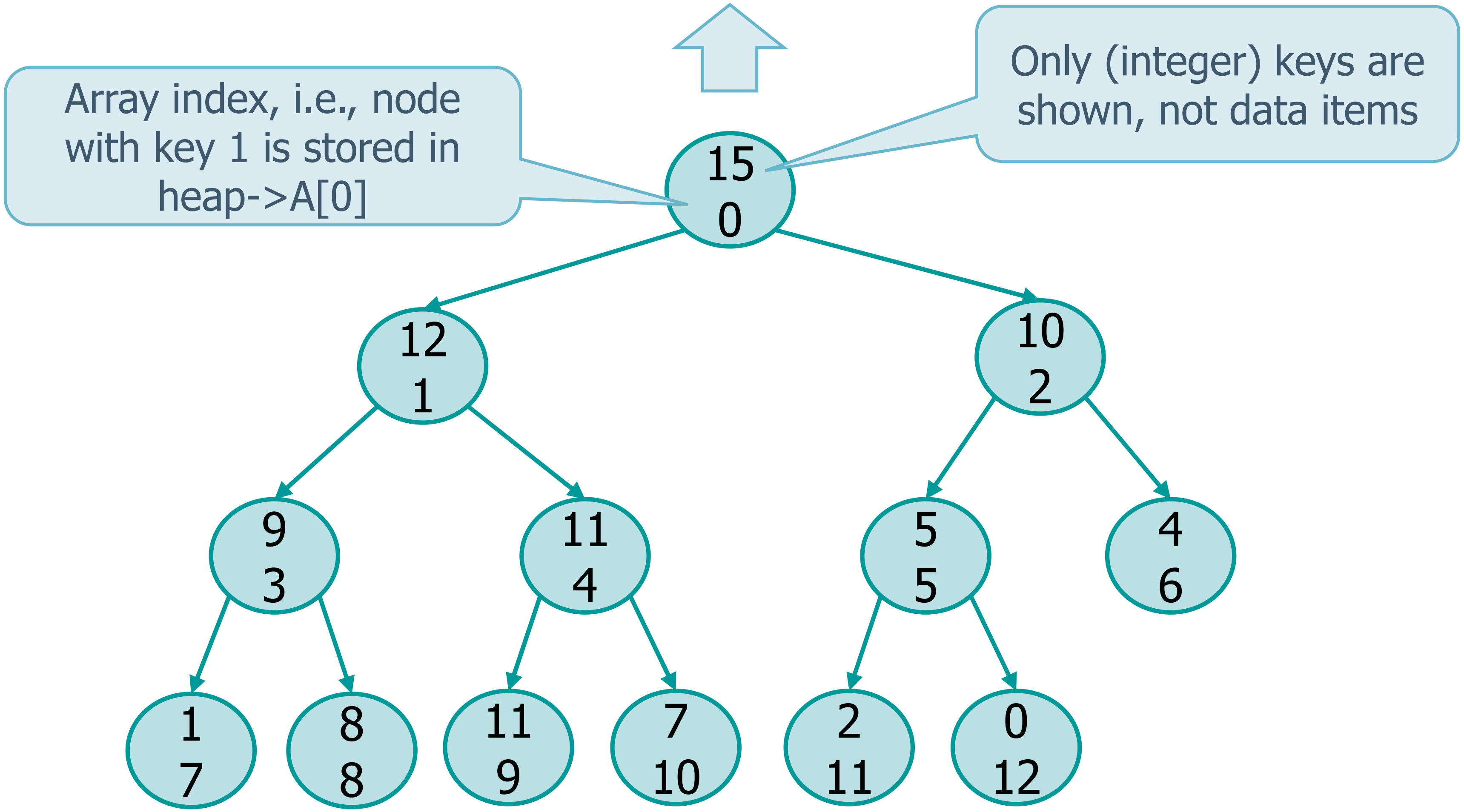
Solution

	0	1	2	3	4	5	6	7	8	9	10	11	12
A	1	15	10	12	11	5	4	9	8	11	7	2	0



Solution

	0	1	2	3	4	5	6	7	8	9	10	11	12
A	15	12	10	9	11	5	4	1	8	11	7	2	0



Implementation

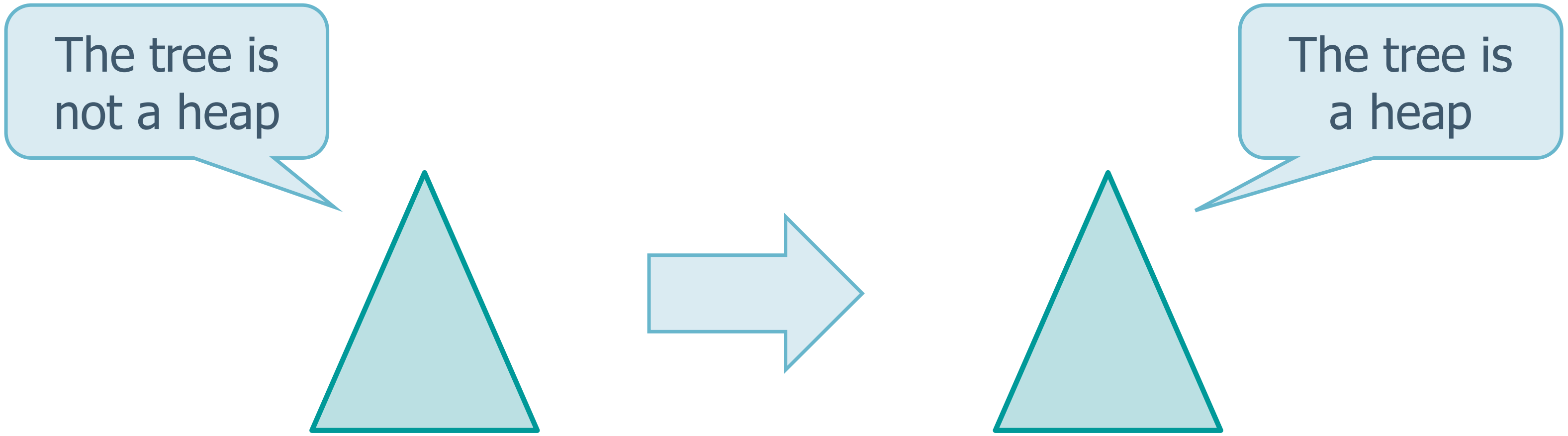
```
void heapify (heap_t heap, int i) {
  int l, r, largest;
  l = LEFT(i);   = 2i+1
  r = RIGHT(i);  = 2i+2
  if ((l<heap->heapsize) &&
      (item_greater (heap->A[l], heap->A[i])))
    largest = l;
  else
    largest = i;
  if ((r<heap->heapsize)&&
      (item_greater (heap->A[r], heap->A[largest])))
    largest = r;
  if (largest != i) {
    swap (heap, i, largest);
    heapify (heap, largest);
  }
  return;
}
```

Function **item_greater** compares keys

idea: switching the nodes if parent less than child,ir

Function heapbuild

- Premises
  - Given a binary tree complete but at the last level and stored into array heap->A
- Outcome
  - Turn the entire array heap->A into a heap



Function heapbuild

- Process
  - Leaves are heaps
  - Apply the **heapify** function
    - Starting from the parent node of the last pair of leaves
    - Move backward on the array until the root is manipulated
- Complexity
  - $T(n) = O(n)$

N calls to heapify should imply O(n·log).
This bound is correct but not tight.
A tighter bound can be proven by a more accurate count of the height of the subtrees and the number of calls to heapify.

Example

- Given the following array, show the result of
  - heapbuild (A)

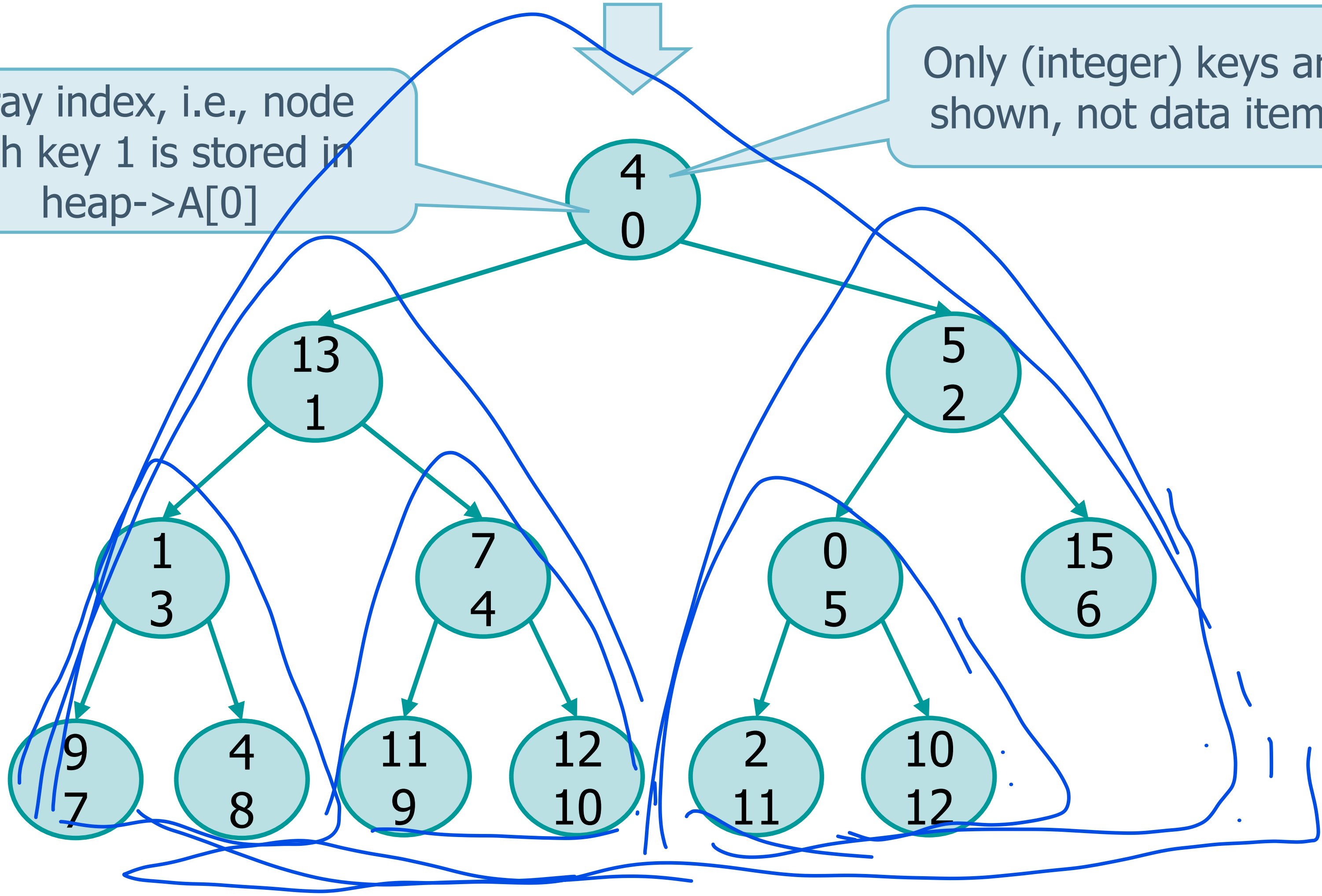
	0	1	2	3	4	5	6	7	8	9	10	11	12
A	4	13	5	1	7	0	15	9	4	11	12	2	10

Solution

	0	1	2	3	4	5	6	7	8	9	10	11	12
A	4	13	5	1	7	0	15	9	4	11	12	2	10

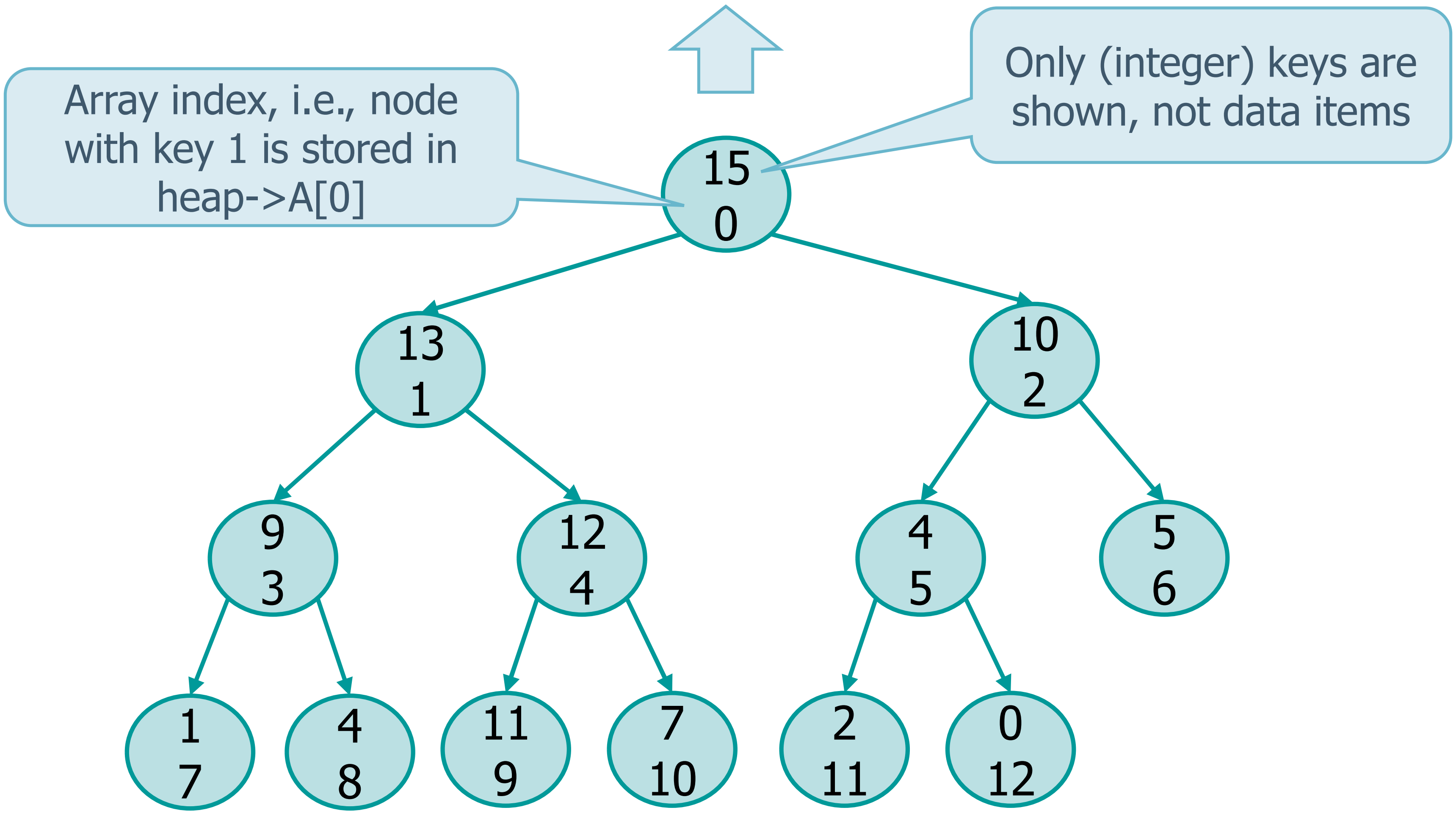
Array index, i.e., node with key 1 is stored in heap->A[0]

Only (integer) keys are shown, not data items



Solution

	0	1	2	3	4	5	6	7	8	9	10	11	12
A	15	13	10	9	12	4	5	1	4	11	7	2	0



Implementation

```
void heapbuild (heap_t heap) {
  int i;

  for (i=(heap->heapsize)/2-1; i >= 0; i--) {
    heapify (heap, i);
  }

  return;
}
```

Start from the last node of the last complete tree level

Call heapify on each node

Move backward untill the root is reached

Function heapsort

- Premises
  - Given a binary tree complete but at the last level and stored into array heap->A
- Outcome
  - Turn array heap->A into a completely sorted array

Function heapsort

- Process
  - Turns the array into a heap using **heapbuild**
  - Swaps first and last elements
  - Decreases heap size by 1
  - Reinforces the heap property using **heapify**
  - Repeats until the heap is empty and the array ordered
- Complexity
  - $T(n) = O(n \cdot log_2 n)$
- In place
- Not stable

A single call to buildheap → O(n)
+
n calls to heapify, each one → O(log n)
=
Implies an overall cost → O(n·logn)

Example

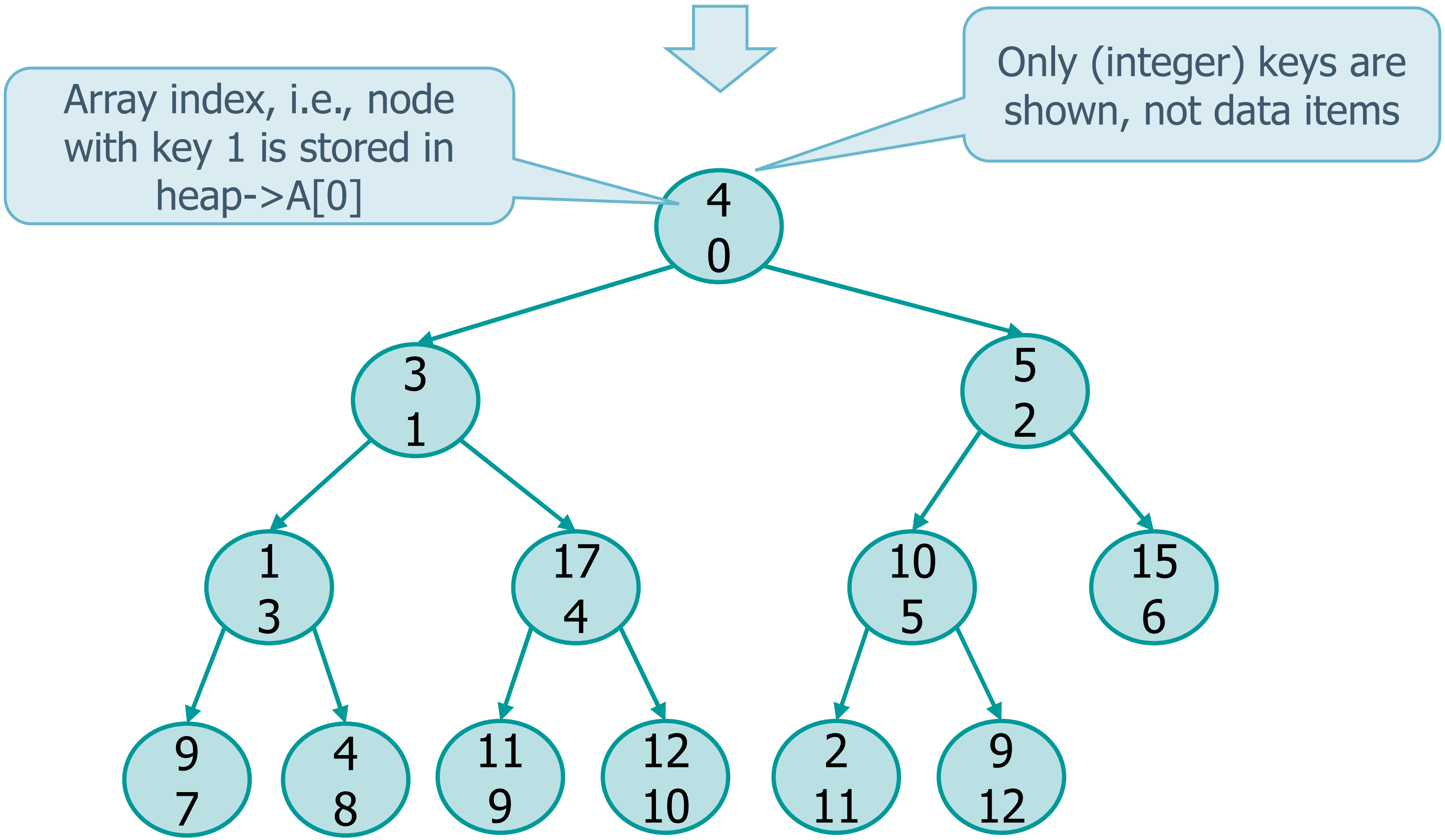
- Given the following array, show the result of
  - heapsort (A)

	0	1	2	3	4	5	6	7	8	9	10	11	12
A	4	3	5	1	17	10	15	9	4	11	12	2	9

Solution

Initial configuration

	0	1	2	3	4	5	6	7	8	9	10	11	12
A	4	3	5	1	17	10	15	9	4	11	12	2	9



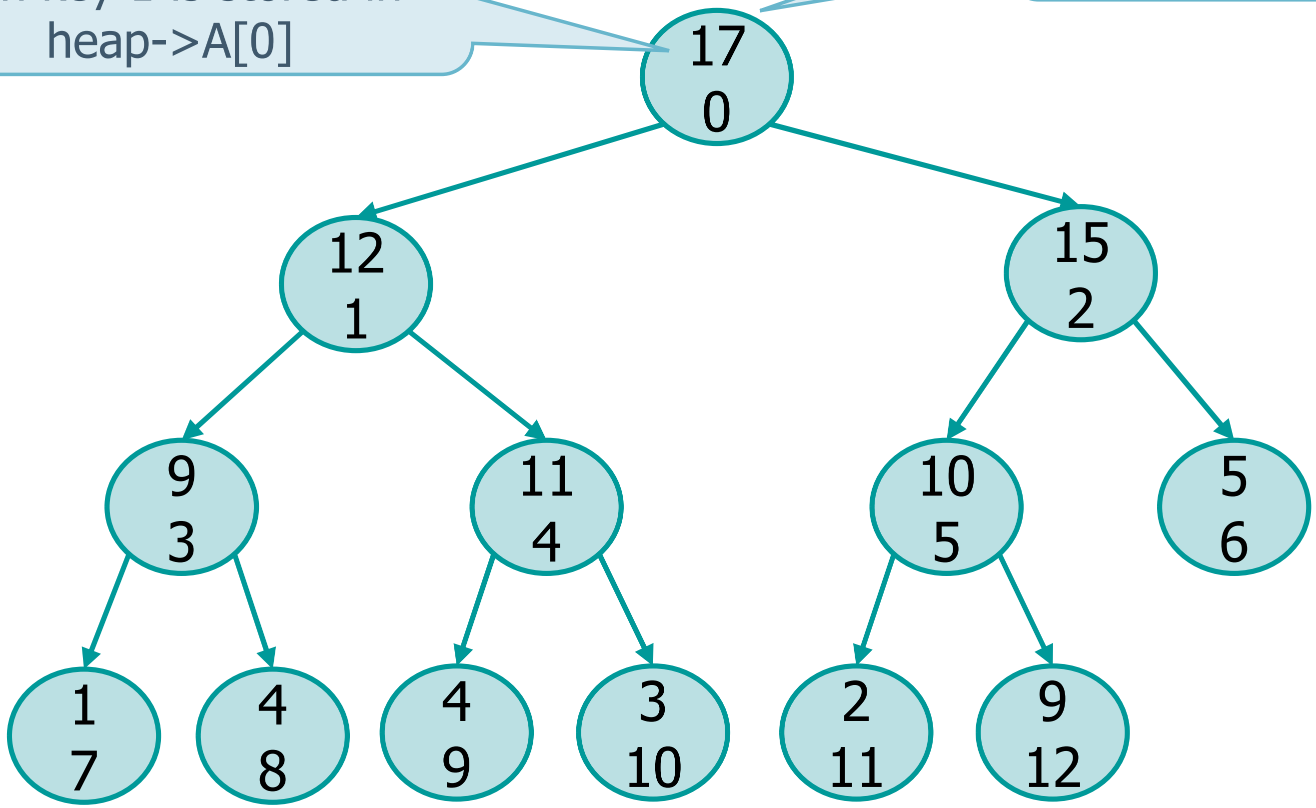
Solution

Configuration after heapbuild

	0	1	2	3	4	5	6	7	8	9	10	11	12
A	17	12	15	9	11	10	5	1	4	4	3	2	9

Only (integer) keys are shown, not data items

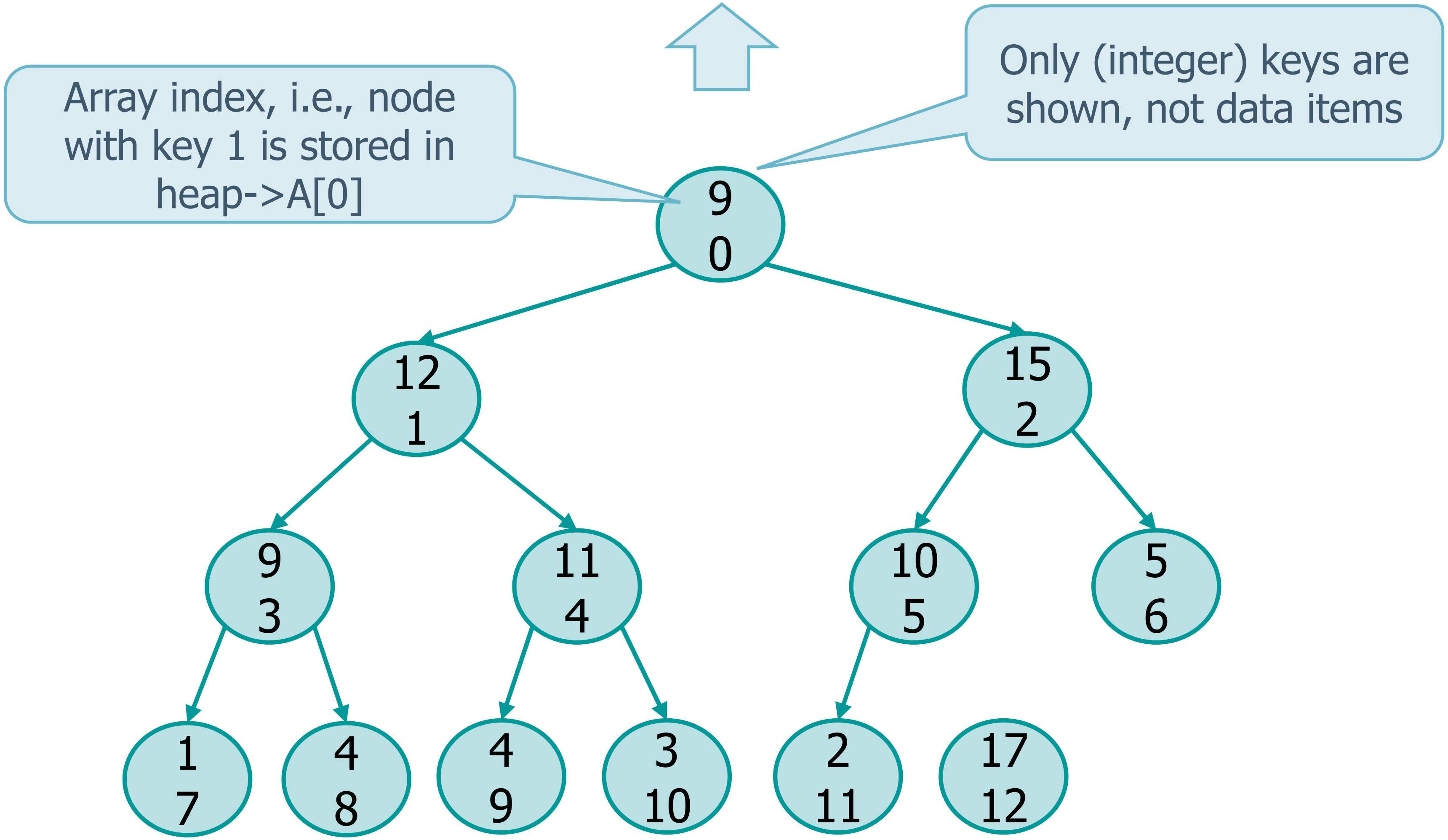
Array index, i.e., node with key 1 is stored in heap->A[0]



Solution

Configuration after swap and heapify

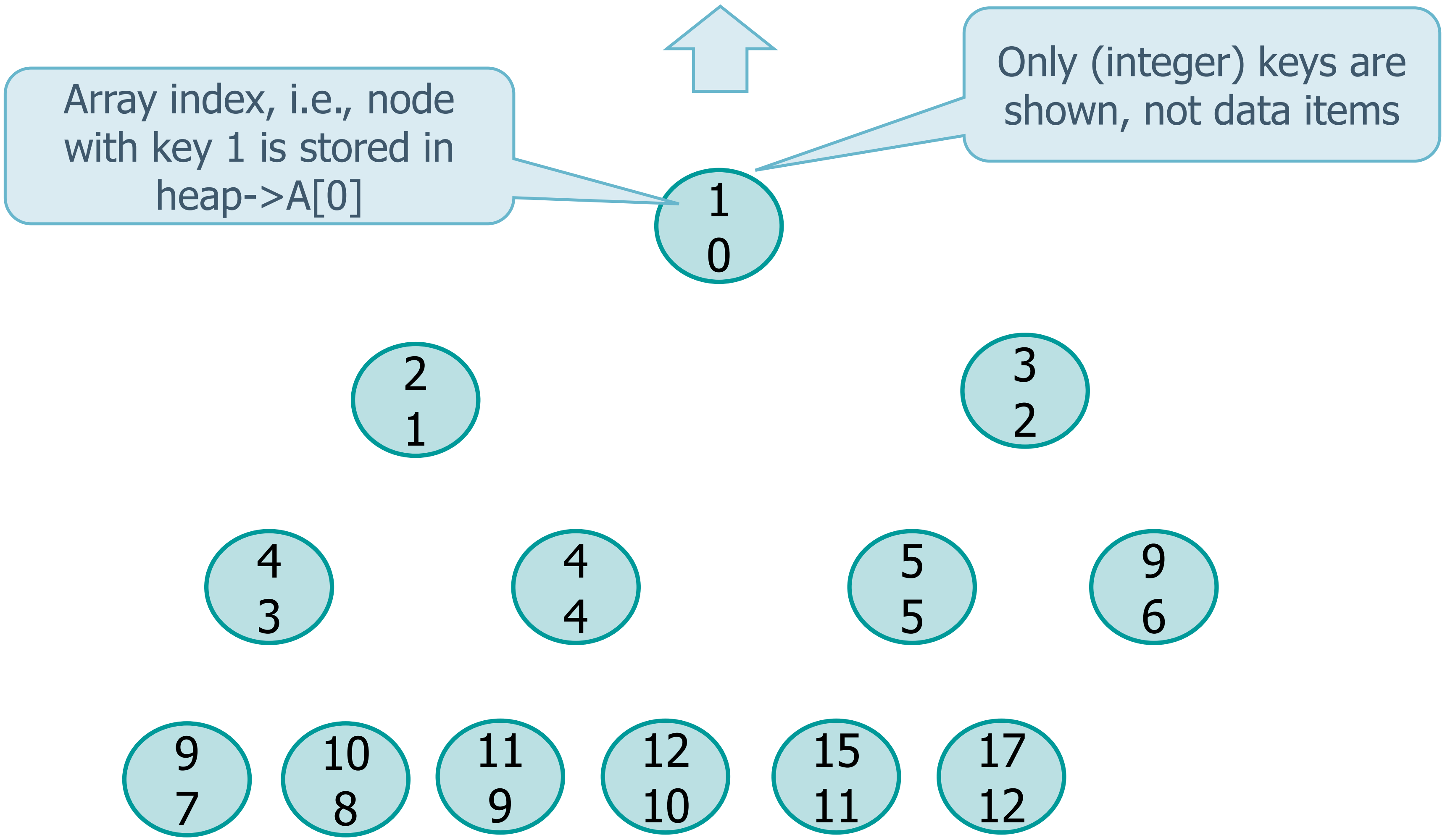
	0	1	2	3	4	5	6	7	8	9	10	11	12
A	17	12	15	9	11	10	5	1	4	4	3	2	9



Solution

Final configuration (result)

	0	1	2	3	4	5	6	7	8	9	10	11	12
A	1	2	3	4	4	5	9	9	10	11	12	15	17



Implementation

```
void heapsort (heap_t heap) {
  int i, tmp;

  heapbuild (heap);

  tmp = heap->heapsize;
  for (i=heap->heapsize-1; i>0; i--) {
    swap (heap, 0, i);
    heap->heapsize--;
    heapify (heap,0);
  }
  heap->heapsize = tmp;

  return;
}
```

Initial heap buld. Forces max value into the root

For heapsize-1 times

Move max value into rigthmost element

Heapify again forcing new max into root

Exercise

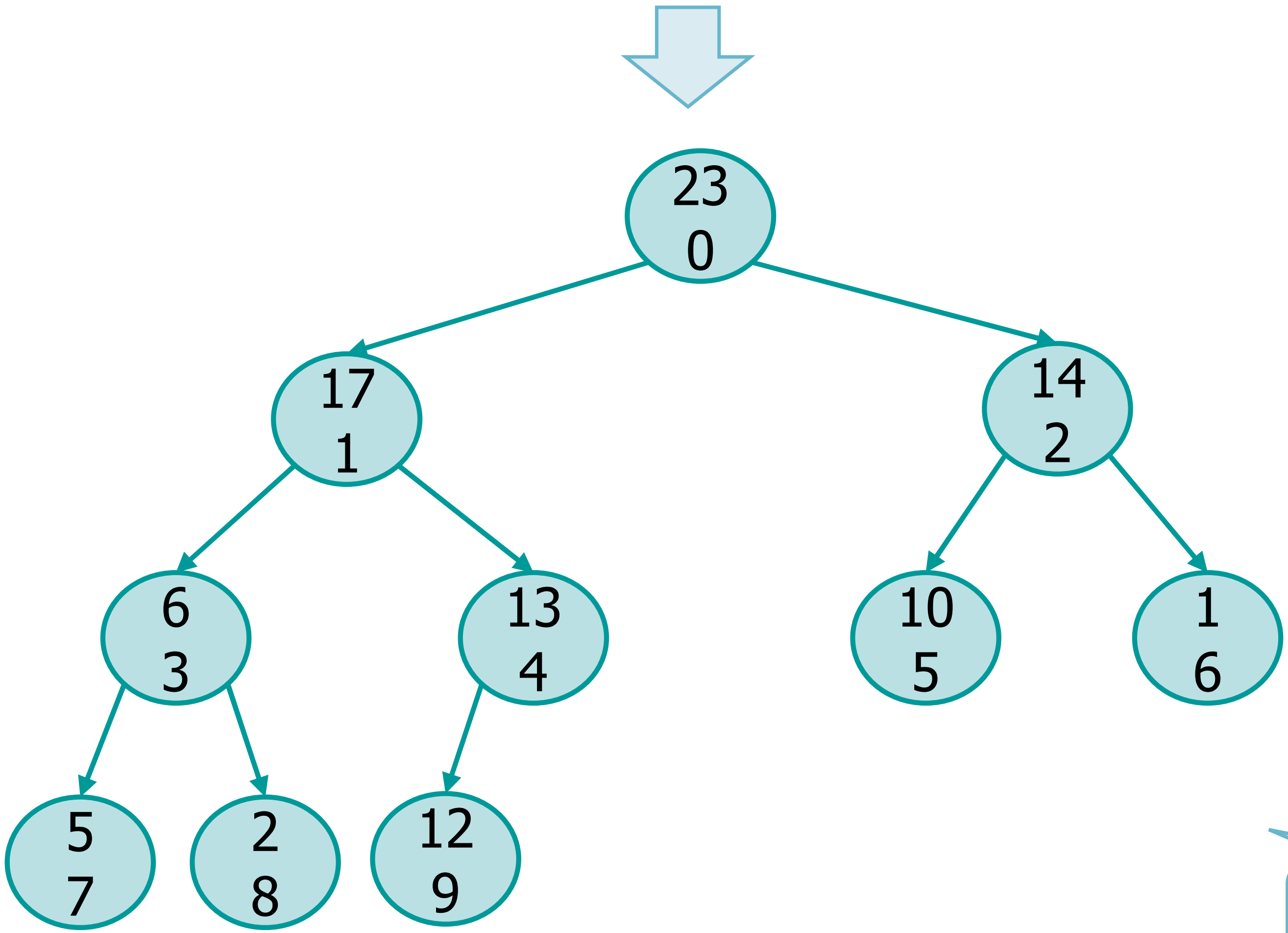
- Are the following sequences a min or a max heap?

	0	1	2	3	4	5	6	7	8	9
A	23	17	14	6	13	10	1	5	2	12

	0	1	2	3	4	5	6	7	8	9	10	11	12
A	4	5	5	11	7	10	23	12	14	8	9	21	19

Solution A

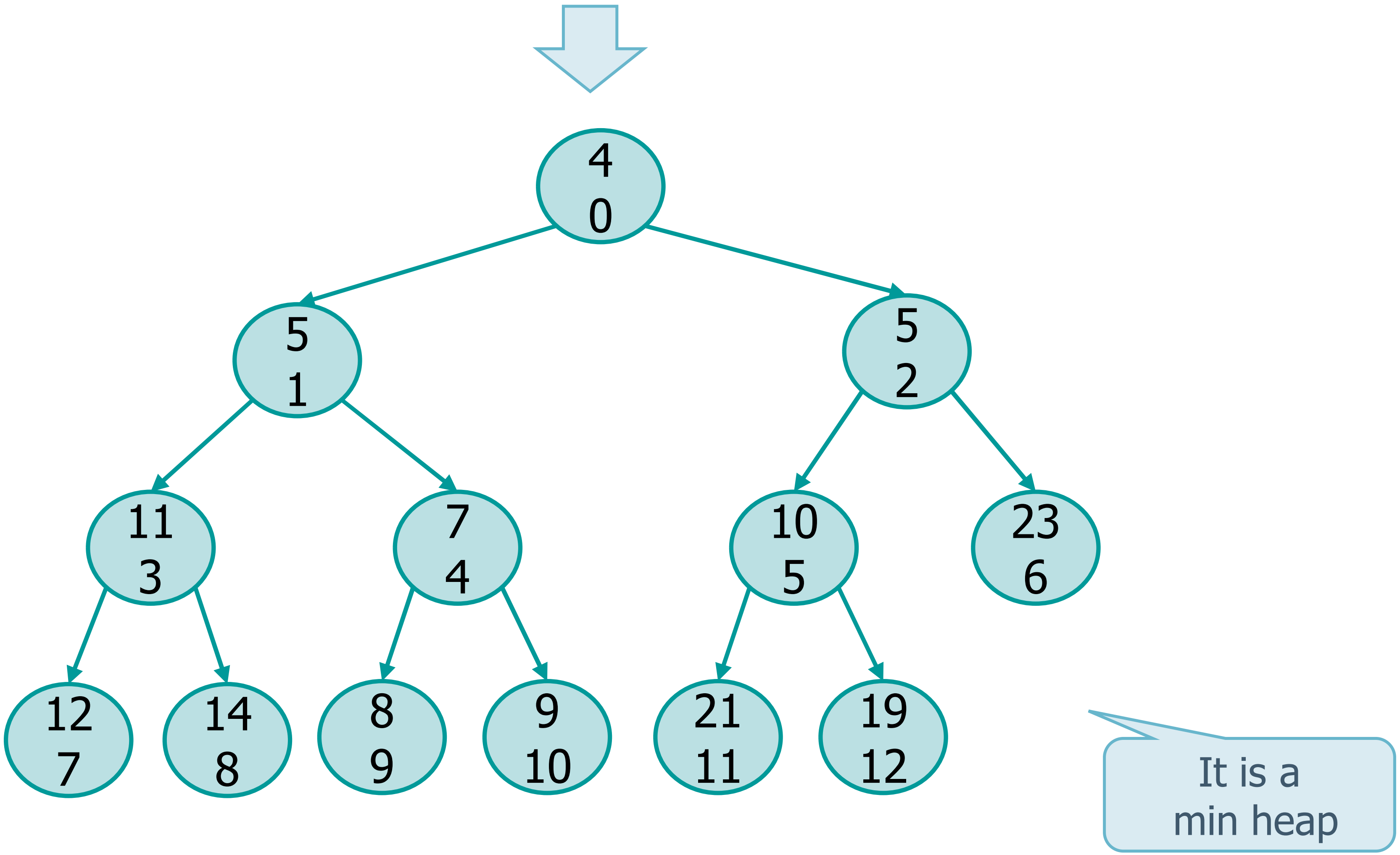
	0	1	2	3	4	5	6	7	8	9
A	23	17	14	6	13	10	1	5	2	12



It is a
max heap

Solution B

	0	1	2	3	4	5	6	7	8	9	10	11	12
A	4	5	5	11	7	10	23	12	14	8	9	21	19



Exercise

- Given the following sequence of integers stored into an array, turn it into a heap and then apply heapsort

	0	1	2	3	4	5	6	7	8	9
A	12	43	8	5	32	9	3	7	11	6

	0	1	2	3	4	5	6	7	8	9	10	11	12
A	12	14	43	10	8	5	61	32	9	38	27	11	56

- Assume that, in the end, the largest (A) or smallest (B) value is stored at the heap's root

Solution A

	0	1	2	3	4	5	6	7	8	9
A	12	43	8	5	32	9	3	7	11	6

Heapbuild

```
No swap               12 43  8  5 32  9  3  7 11  6
[3<->8]               12 43  8 11 32  9  3  7  5  6
[2<->5]               12 43  9 11 32  8  3  7  5  6
No swap               12 43  9 11 32  8  3  7  5  6
[0<->1][1<->4]        43 32  9 11 12  8  3  7  5  6
```

Heapsort

```
[0<->1][1<->4]          32 12  9 11  6  8  3  7  5 43
[0<->1][1<->3][3<->7]   12 11  9  7  6  8  3  5 32 43
[0<->1][1<->3]          11  7  9  5  6  8  3 12 32 43
[0<->2][2<->5]           9  7  8  5  6  3 11 12 32 43
[0<->2]                  8  7  3  5  6  9 11 12 32 43
[0<->1]                  7  6  3  5  8  9 11 12 32 43
[0<->1]                  6  5  3  7  8  9 11 12 32 43
[0<->1]                  5  3  6  7  8  9 11 12 32 43
No swap                  3  5  6  7  8  9 11 12 32 43
```

Solution B

	0	1	2	3	4	5	6	7	8	9	10	11	12
A	12	14	43	10	8	5	61	32	9	38	27	11	56

Heapbuild

No swap	12	14	43	10	8	5	61	32	9	38	27	11	56
No swap	12	14	43	10	8	5	61	32	9	38	27	11	56
[3<->8]	12	14	43	9	8	5	61	32	10	38	27	11	56
[2<->5][5<->11]	12	14	5	9	8	11	61	32	10	38	27	43	56
[1<->4]	12	8	5	9	14	11	61	32	10	38	27	43	56
[0<->2][2<->5]	5	8	11	9	14	12	61	32	10	38	27	43	56

Heapsort

[0<->1][1<->3][3<->8]	8	9	11	10	14	12	61	32	56	38	27	43	5
[0<->1][1<->3][3<->7]	9	10	11	32	14	12	61	43	56	38	27	8	5
[0<->1][1<->4]	10	14	11	32	27	12	61	43	56	38	9	8	5
[0<->2][2<->5]	11	14	12	32	27	38	61	43	56	10	9	8	5
[0<->2][2<->5]	12	14	38	32	27	56	61	43	11	10	9	8	5
[0<->1][1<->4]	14	27	38	32	43	56	61	12	11	10	9	8	5
[0<->1][1<->3]	27	32	38	61	43	56	14	12	11	10	9	8	5
[0<->1][1<->4]	32	43	38	61	56	27	14	12	11	10	9	8	5
[0<->2]	38	43	56	61	32	27	14	12	11	10	9	8	5
[0<->1]	43	61	56	38	32	27	14	12	11	10	9	8	5
No swap	56	61	43	38	32	27	14	12	11	10	9	8	5
No swap	61	56	43	38	32	27	14	12	11	10	9	8	5