Heap

Heap Sort

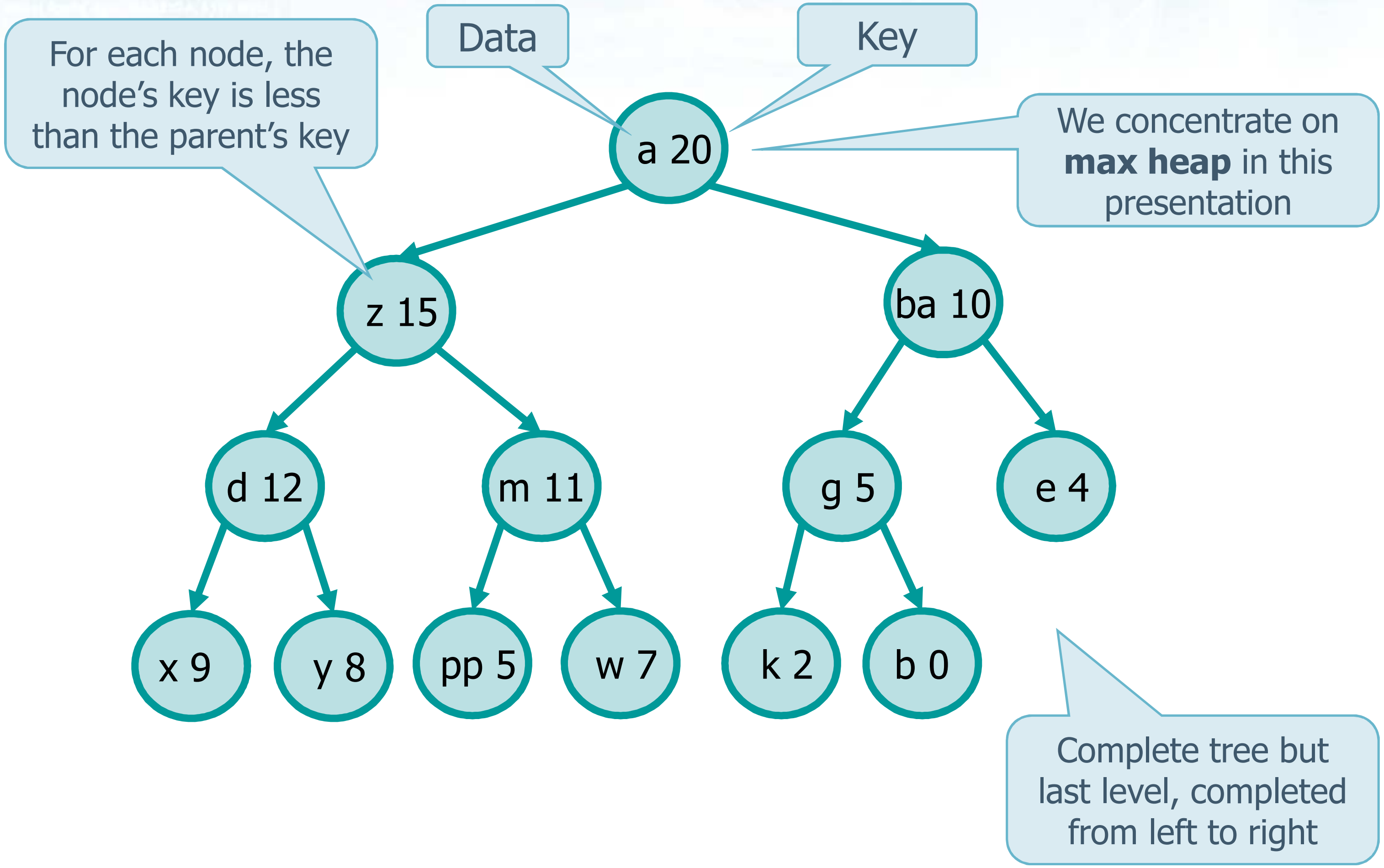
Paolo Camurati and Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

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 (**max** heap)
    - For each node different from the root we have that the key of the node is less than the key of the parent node
      - key[parent(node)] ≥ key(node)
- Consequence
  - The maximum key is in the root

We have both **max** and **min** heaps

Example


For each node, the node's key is less than the parent's key
Data
Key
We concentrate on **max heap** in this presentation
a 20
z 15
ba 10
d 12
m 11
g 5
e 4
x 9
y 8
pp 5
w 7
k 2
b 0
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)

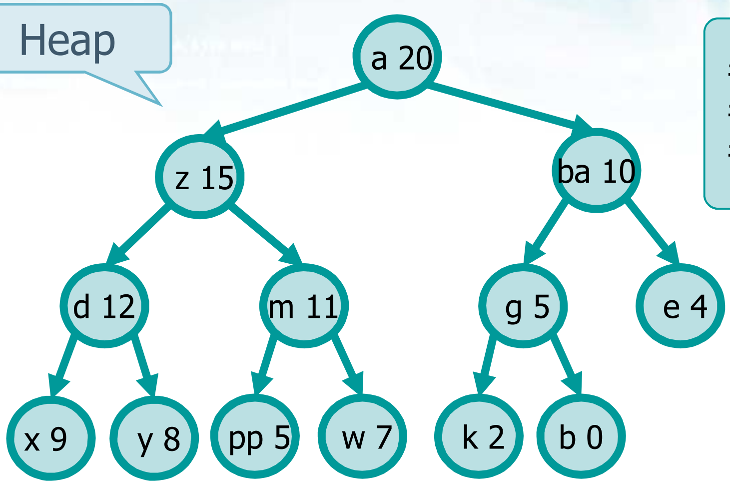
Heapsize specifiy the humber of elements stored in the heap heap->A

ADT Heap

- The root of the heap is stored in
  - heap->A[0]
- Given a node i, we define
  - LEFT(i)= 2·i+1
  - RIGHT(i) = 2·i+2
  - PARENT(i)=(i-1)/2
- Thus given a node heap->A[i]
  - Its left child is heap->A[LEFT(i)]
  - Its right child is heap->A[RIGHT(i)]
  - Its parent is heap->A[PARENT(i)]

Example

Heap




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

Array representation

heap->A

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		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

heap->heapsize = 13

Array (maximum)
maxN = 15

Heap sort

- Focusing of the task of sorting, the heap sort ordering algorithm, is implemented through 3 functions
  - heapify (heap, i)
  - heapbuild (heap)
  - heapsort (heap)
- These functions call each other to elegantly build-up the final ordering

Function heapify

- Premises
  - Given a 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)

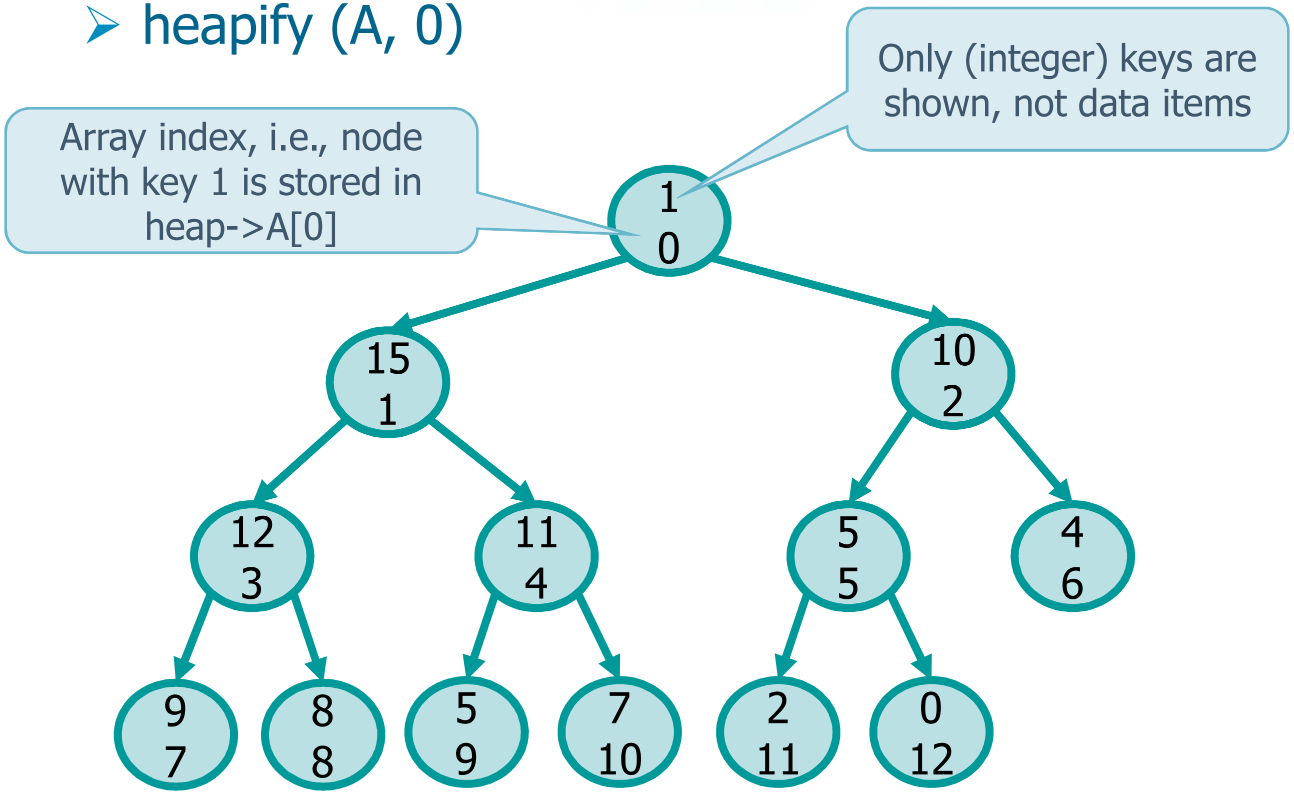
Function heapify

- 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(lg n)

Height of the node
log n for the entire tree

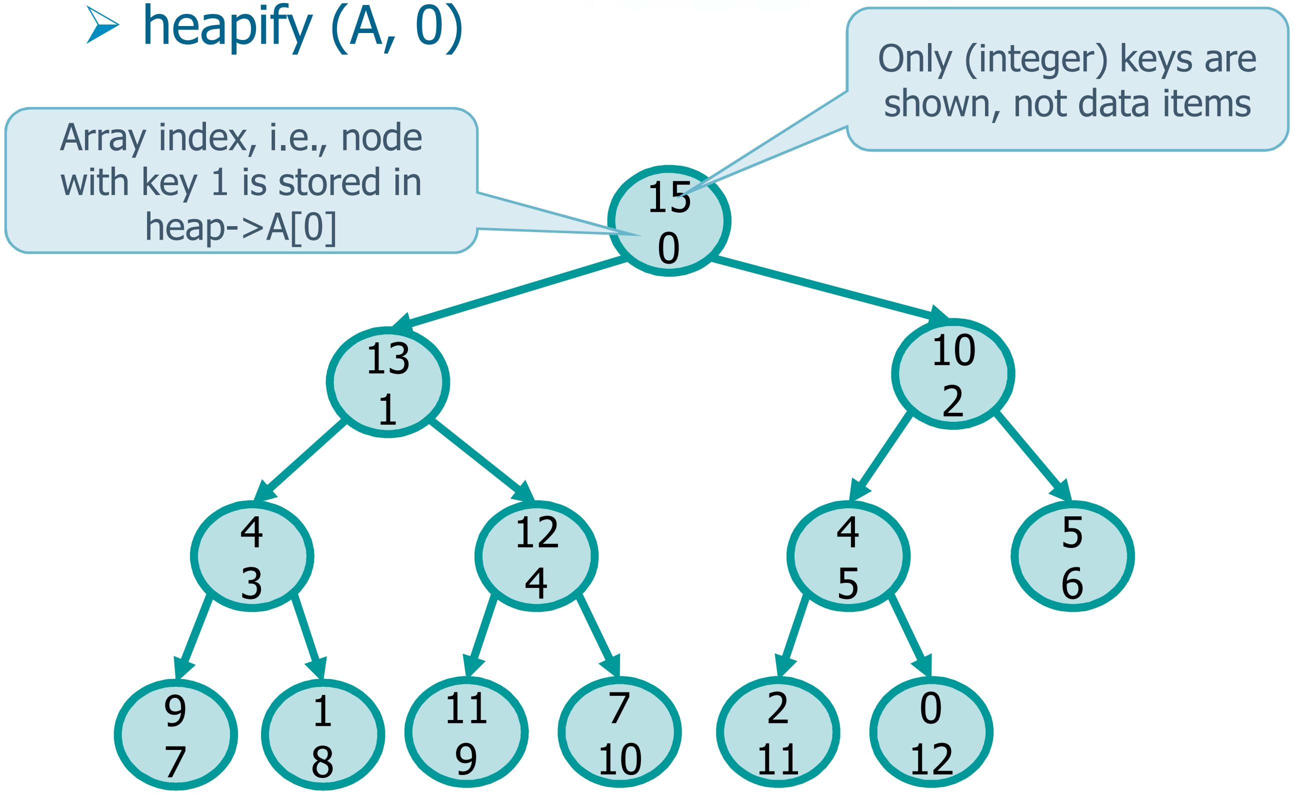
Example

- Call function
  - heapify (A, 0)



Solution

- Call function
  - heapify (A, 0)



Implementation

```
void heapify (heap_t heap, int i) {
  int l, r, largest;
  l = LEFT(i);
  r = RIGHT(i);
  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

Function heapbuild

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

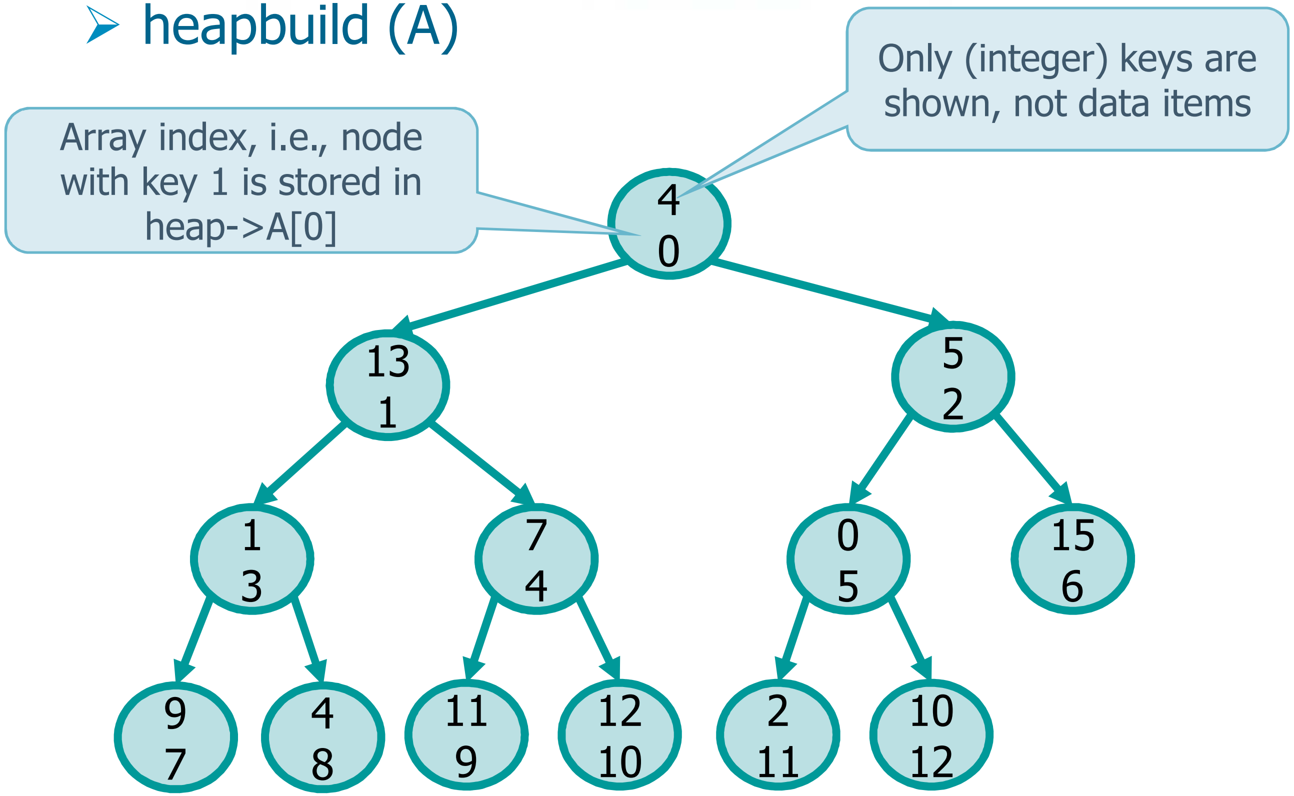
Function heapbuild

- 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.

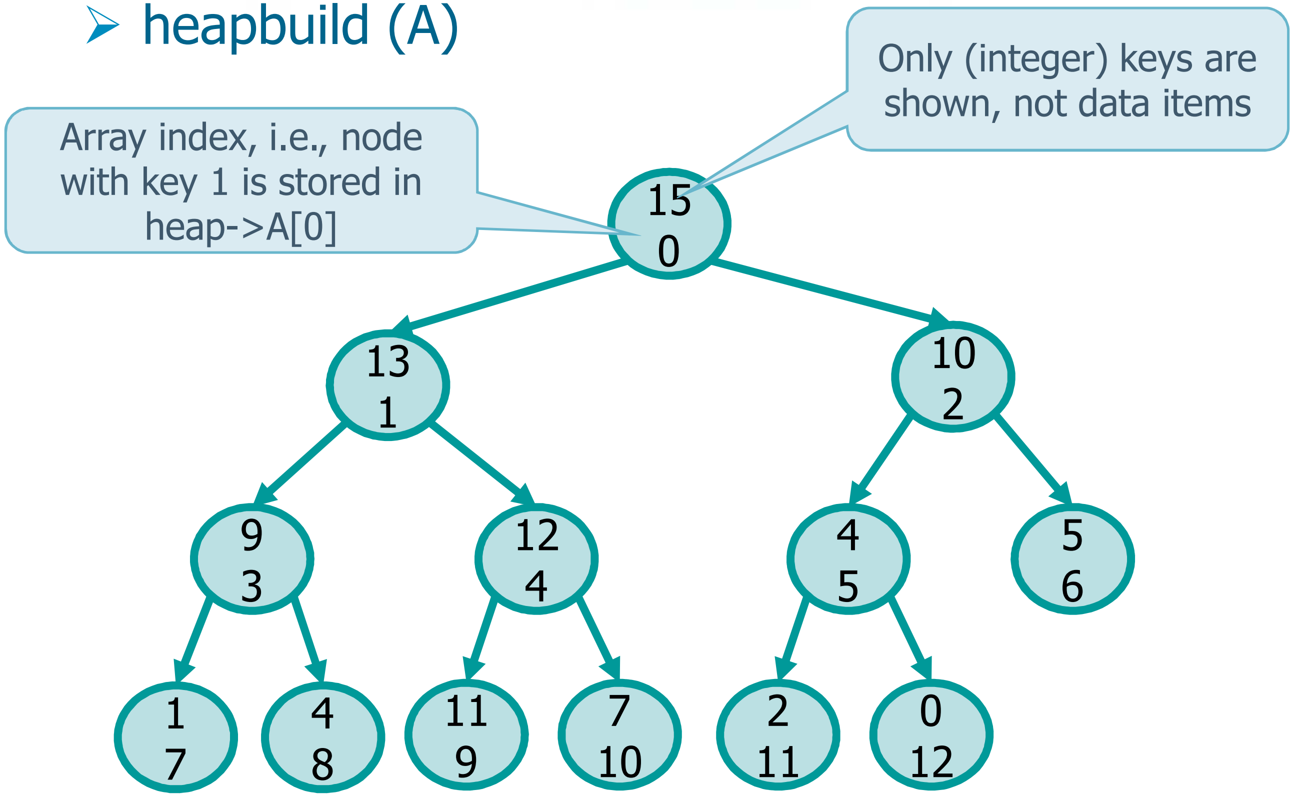
Exercise

- Call function
  - heapbuild (A)



Solution

- Call function
  - heapbuild (A)



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 leve

Call heapify on each node

Move backward till the root

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

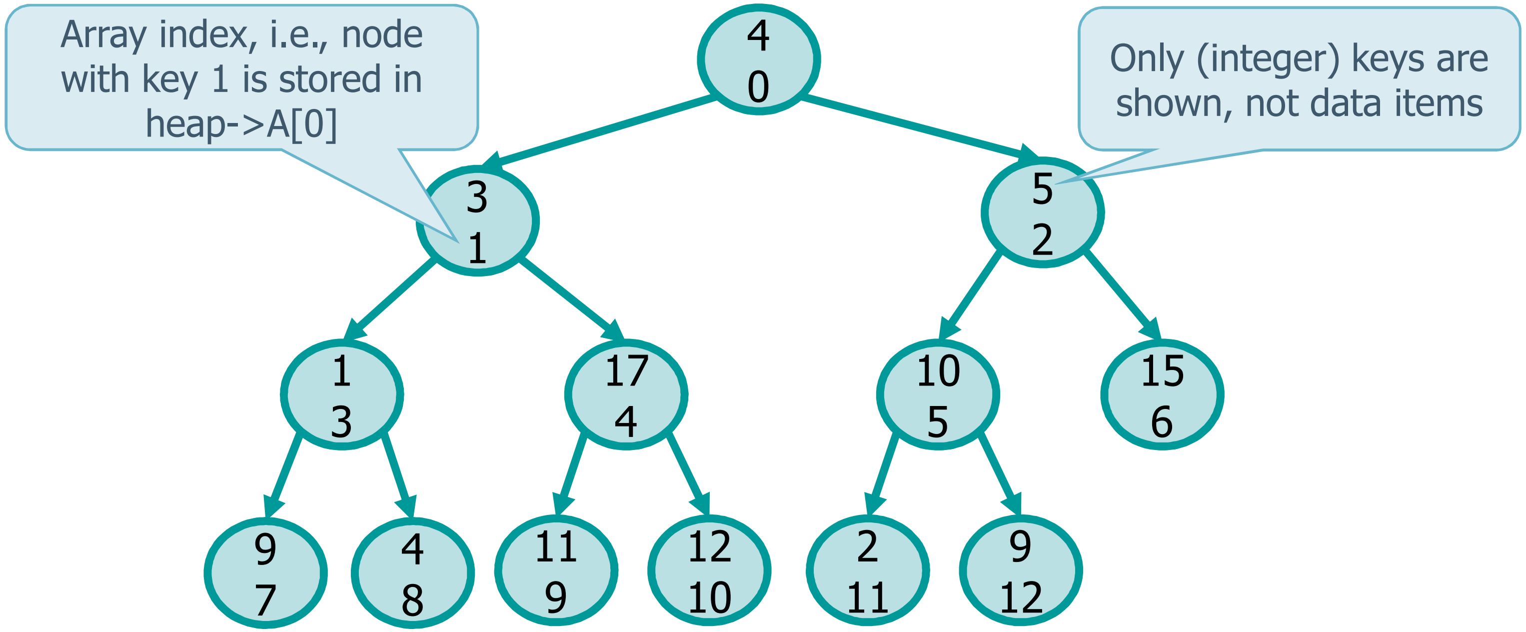
- Process
  - Turns the array into a heap using **heapbuild**
  - Swaps first and last elements
  - Decreases heap size by 1
  - Reinforces the heap property using **heapify**
  - Repeats until the heap is empty and the array ordered
- Complexity
  - T(n)= O (n · lg n)
- In place
- Not stable

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

Exercise

- Call function
  - heapsort (A)

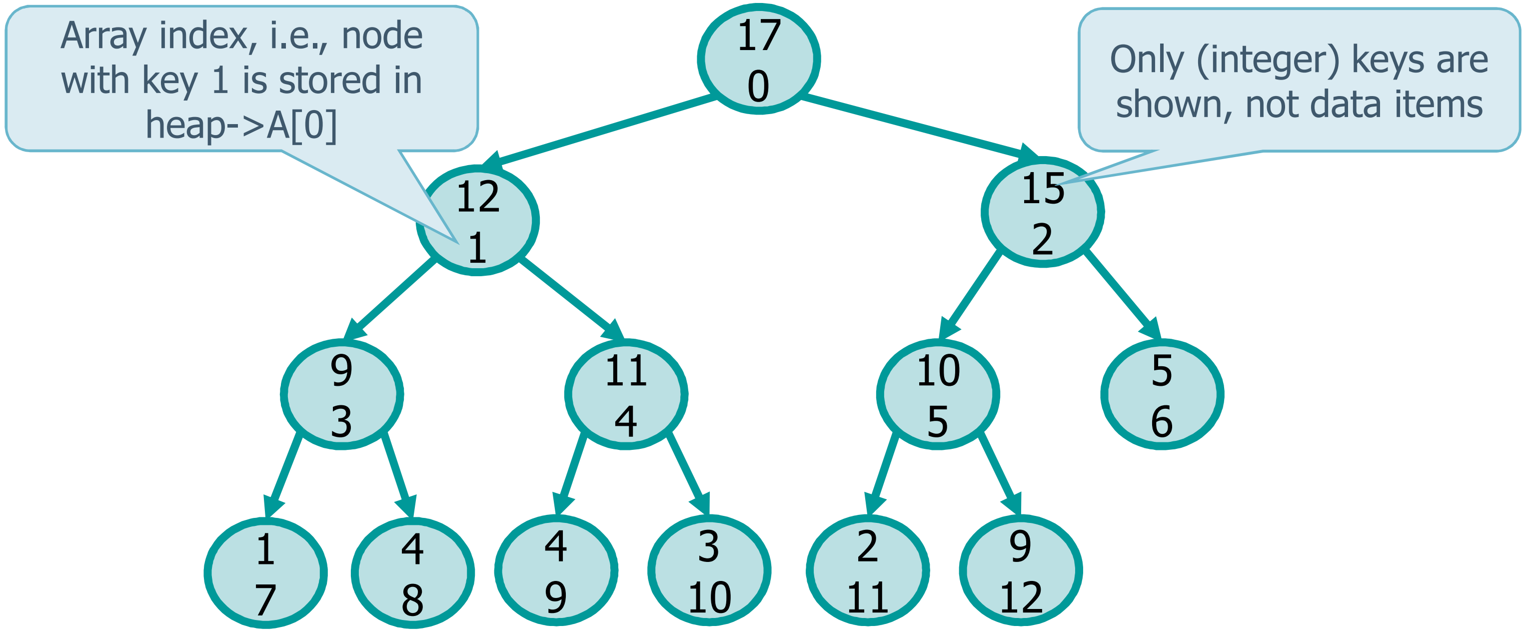
	0	1	2	3	4	5	6	7	8	9	10	11	12
A	4	3	5	1	17	10	15	9	4	11	12	2	9



Solution

- Call function
  - heapsort (A)

	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

- Is the following sequence a max heap?
  - 23 17 14 6 13 10 1 5 7 12

Exercise

- Sort the following sequence in ascending order using heap-sort
  - 12 14 43 10 80 100 61 32 89 78 44 57 11 68 85 56

Exercise

- Sort the following sequence in descending order using heap-sort
  - 41 58 65 36 12 69 13 14 23 10 60 100 78 44 17 21