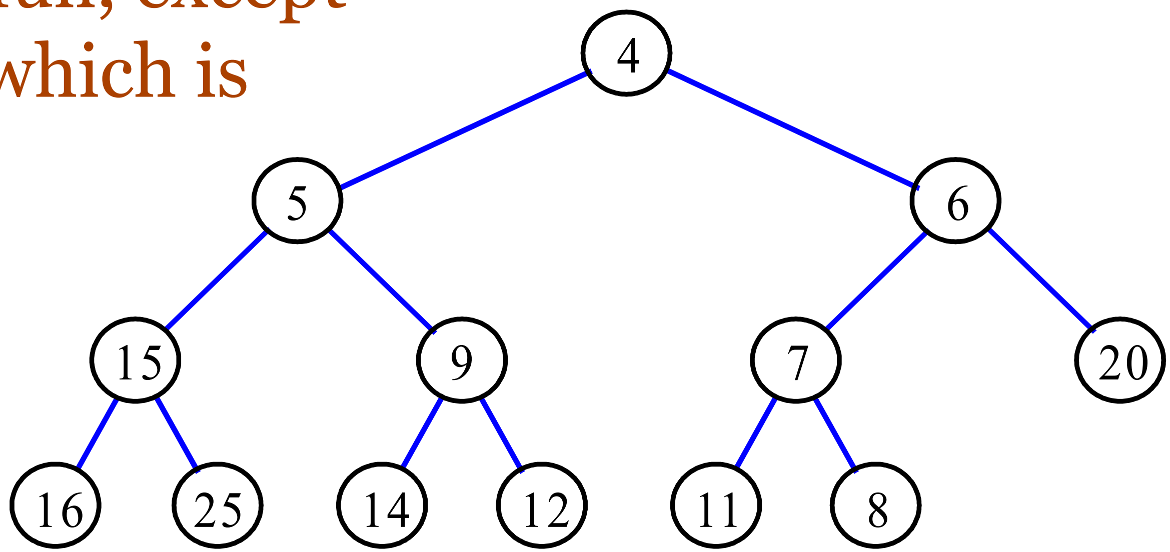
CSCI 340
Data Structures and Algorithm Analysis

Heaps

Thanks to Dr. Rada Mihalcea of the University of North Texas for sharing her slides

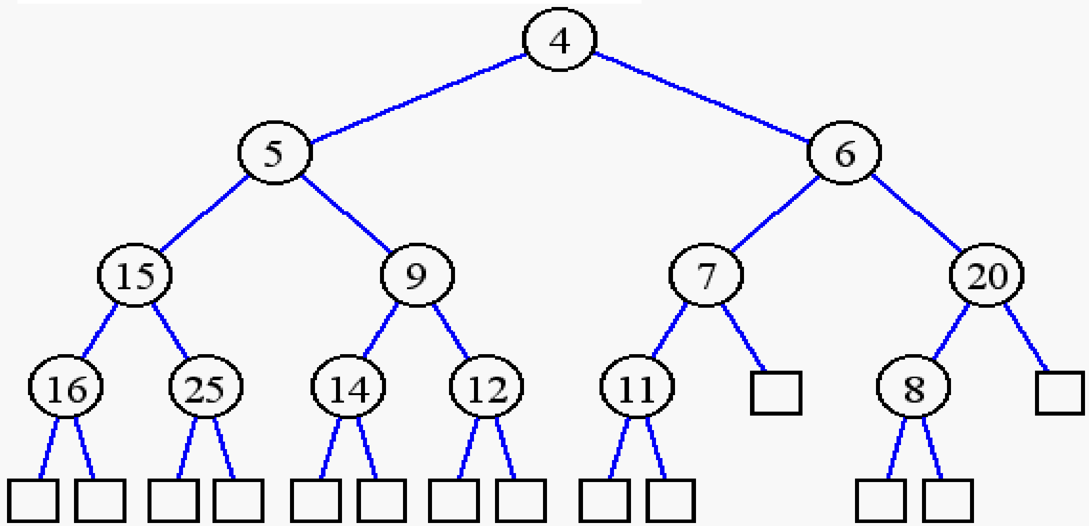
Heaps

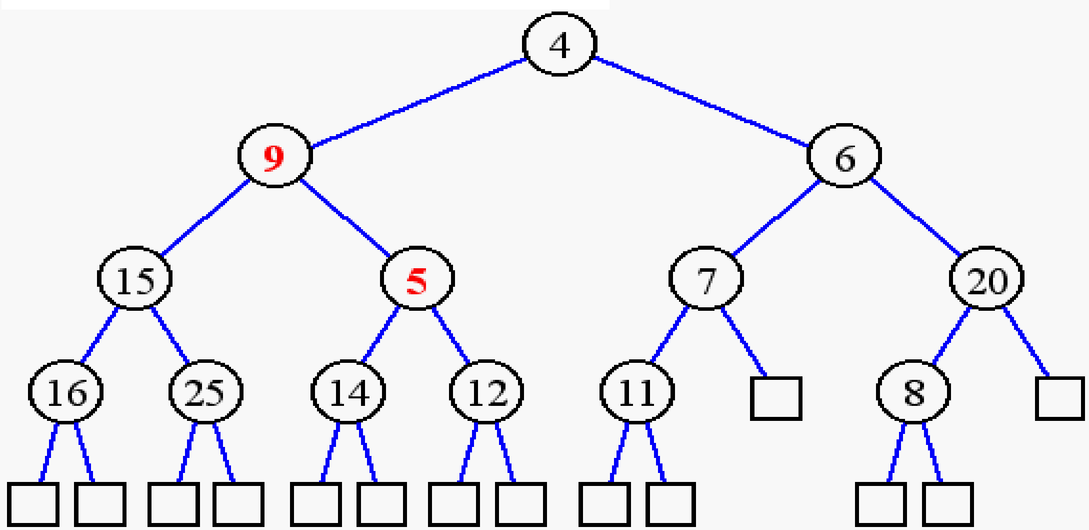
- A *heap* is a binary tree that stores key-element pairs at its nodes and satisfies two properties:
  - **Min-Heap: key(parent) ≤ key(child)**
    [OR Max-Heap: key(parent) >= key(child)]
  - All levels are full, except the last one, which is left-filled



- This set of slides uses min-heap for exampels.

Heap or Not a Heap?



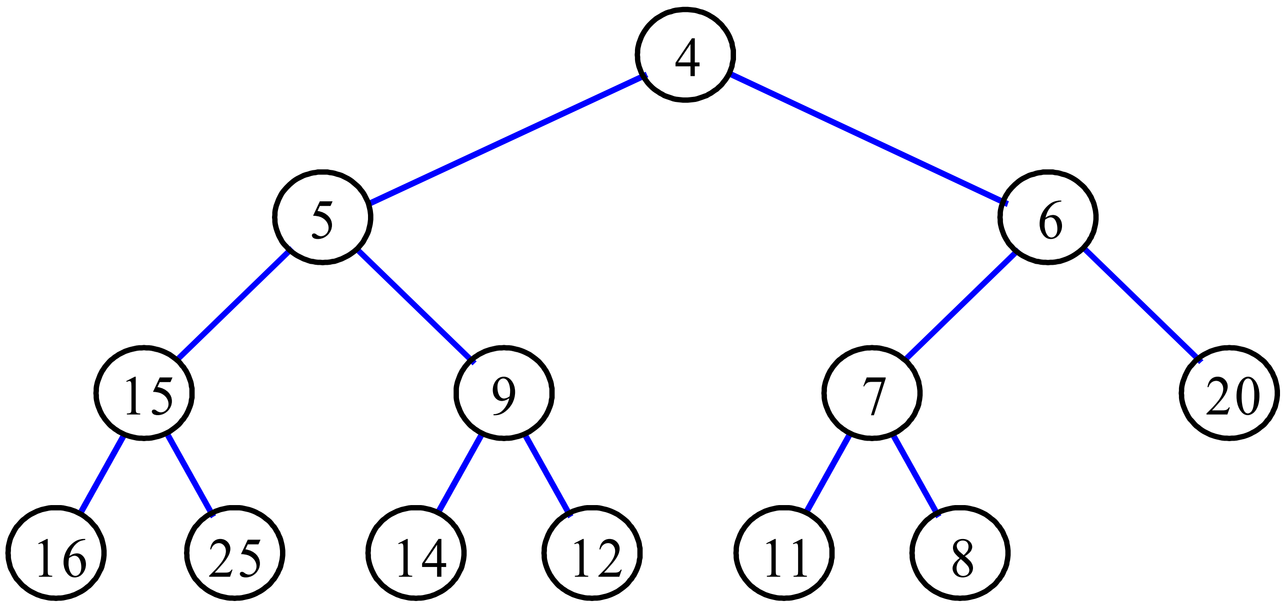


What are Heaps Useful for?

- To implement priority queues
- Priority queue = a queue where all elements have a “priority” associated with them
- Remove in a priority queue removes the element with the smallest priority
  - insert
  - removeMin
- HeapSort
  - In-place
  - Effecient

Heap Properties

- A heap storing n keys has height $h = \lfloor \log n \rfloor$, which is $O(\log n)$



Abstract Data Type for Min Heap

objects: n > 0 elements organized in a binary tree so that the value in each node is at least as large as those in its children

method:

Heap Create(MAX_SIZE)::= create an empty heap that can hold a maximum of max_size elements

Boolean HeapFull(heap, n)::= if (n==max_size) return TRUE
else return FALSE

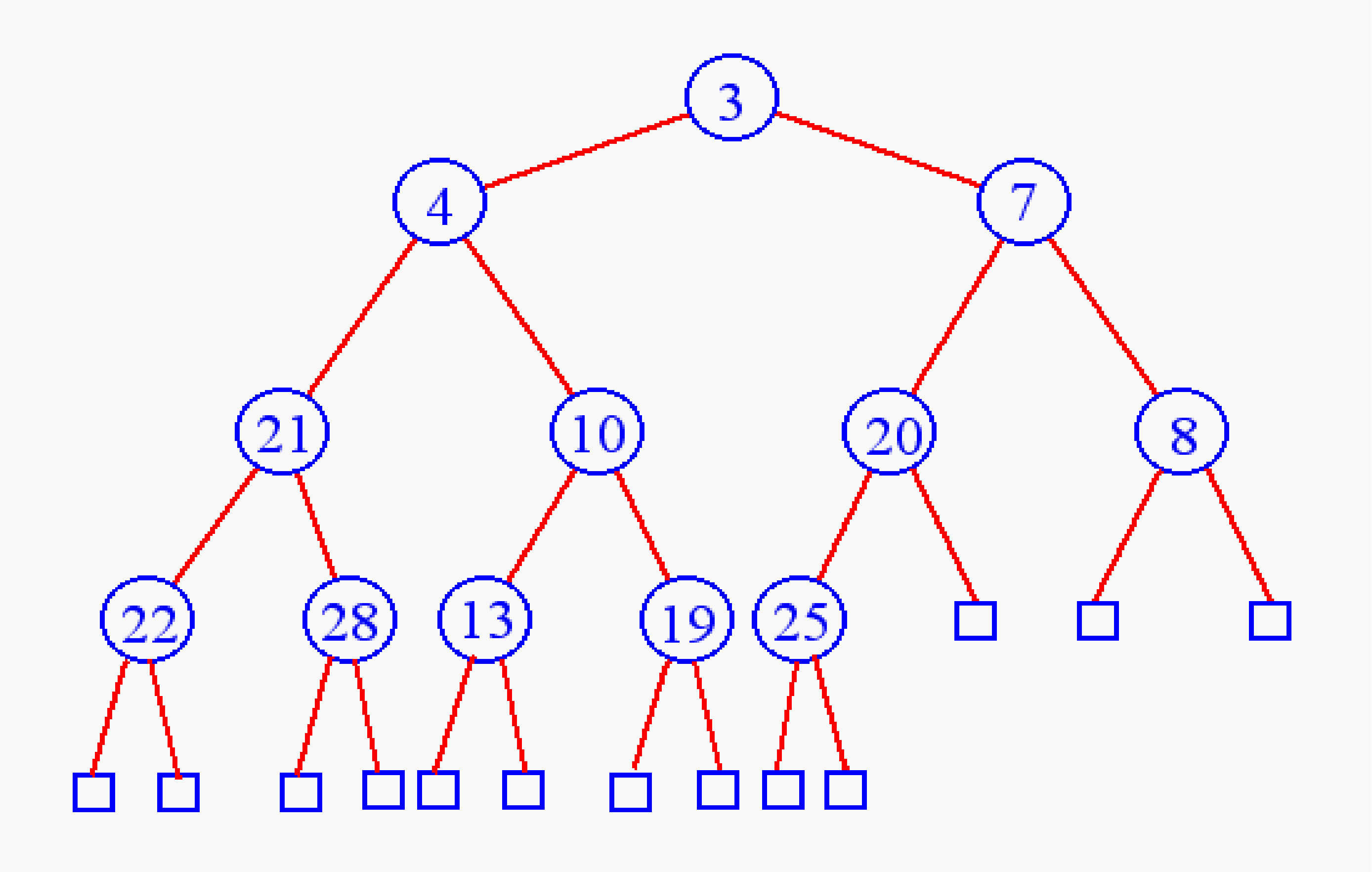
Heap Insert(heap, item, n)::= if (!HeapFull(heap,n)) insert item into heap and return the resulting heap
else return error

Boolean HeapEmpty(heap, n)::= if (n>0) return FALSE
else return TRUE

Element Delete(heap,n)::= if (!HeapEmpty(heap,n)) return one instance of the smallest element in the heap and remove it from the heap
else return error

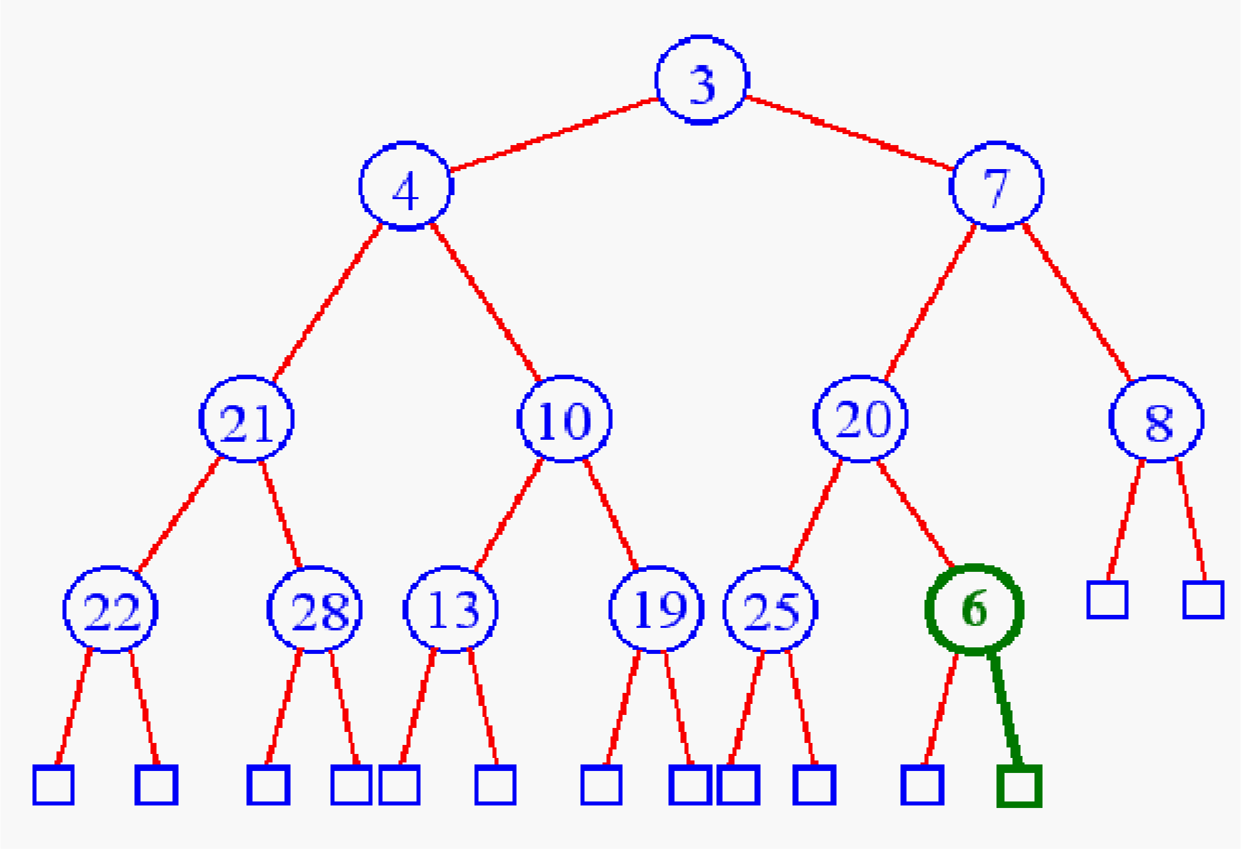
Heap Insertion

- Insert 6



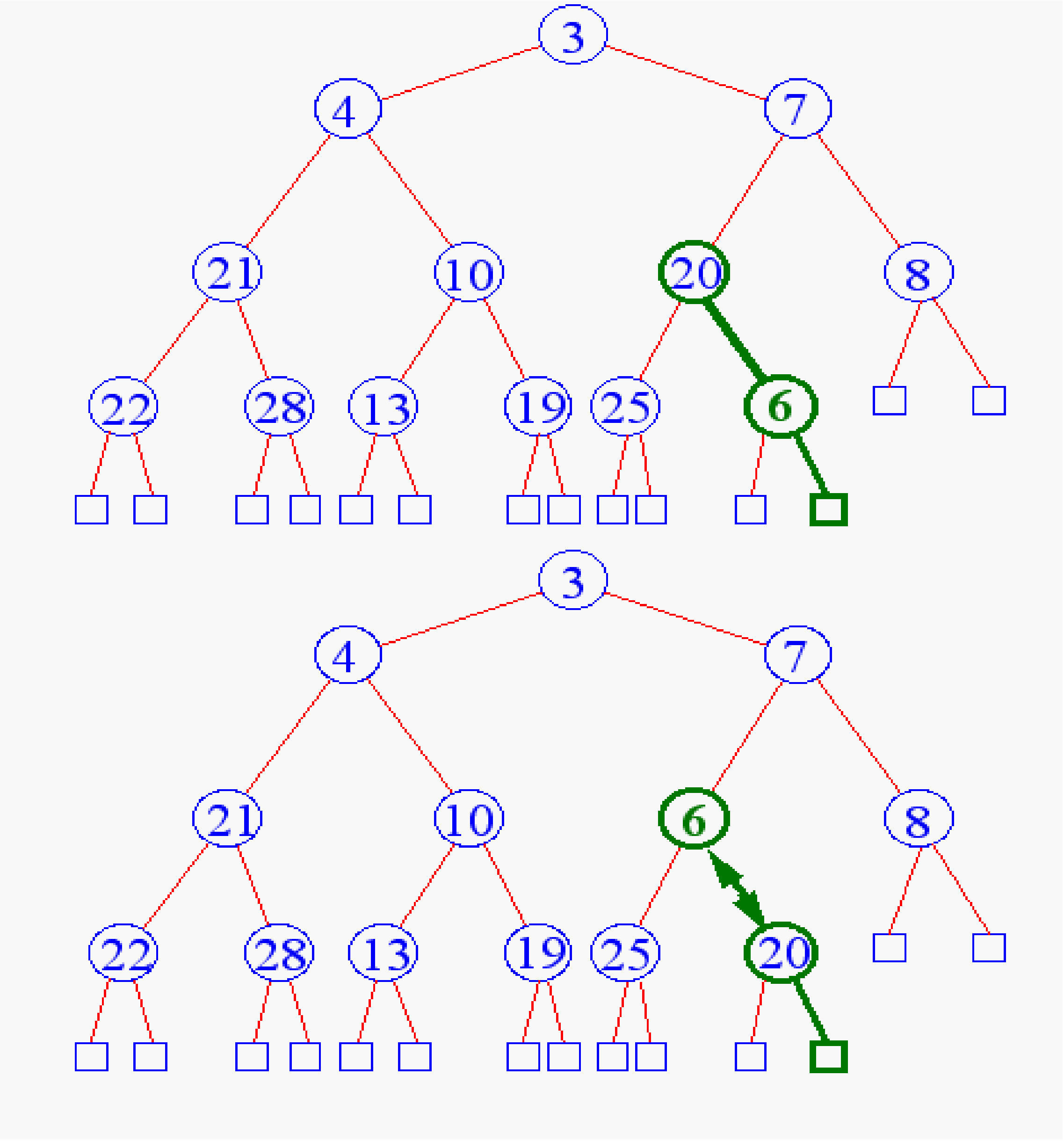
Heap Insertion

- Add key in next available position

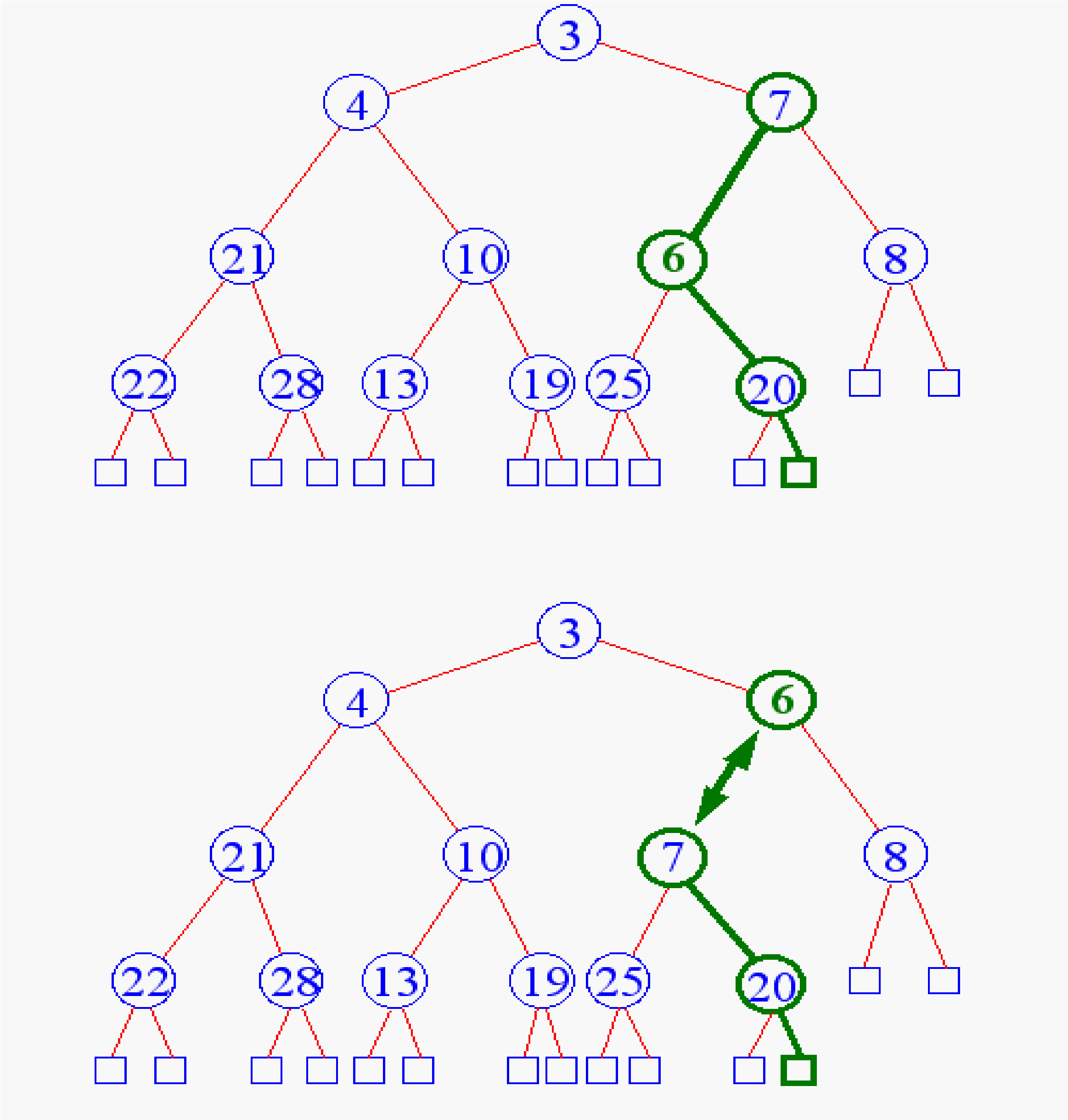


Heap Insertion

- Begin Upheap

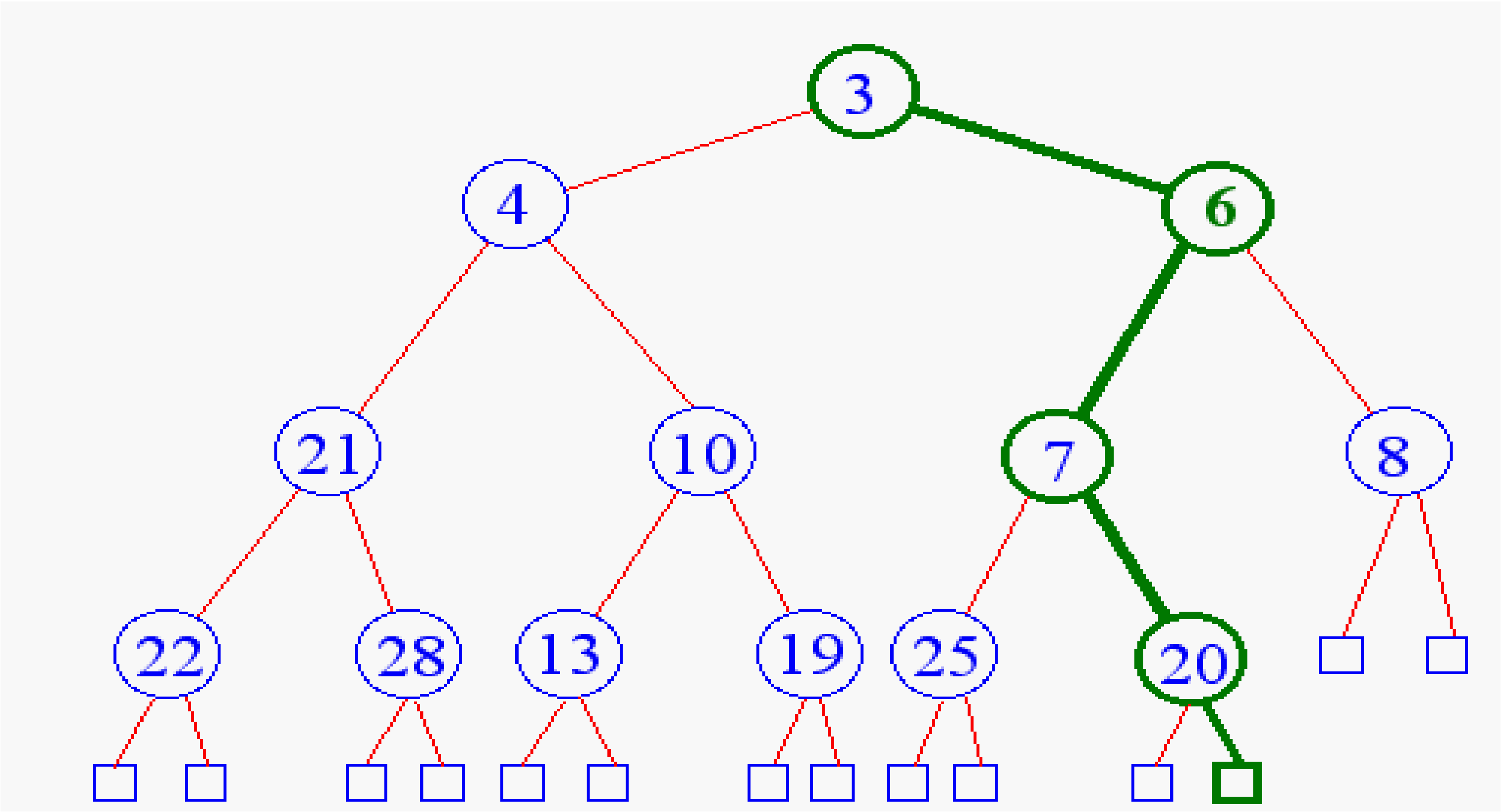


Heap Insertion



Heap Insertion

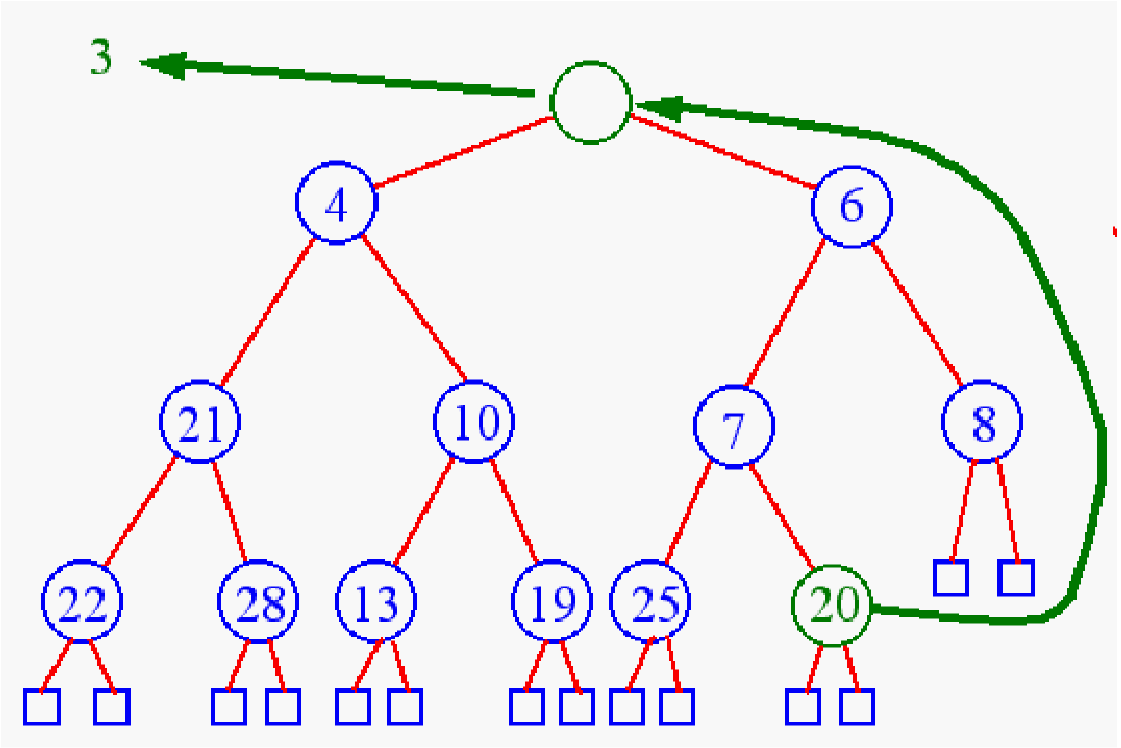
- Terminate upheap when
  - reach root
  - key child is greater than key parent



Heap Removal

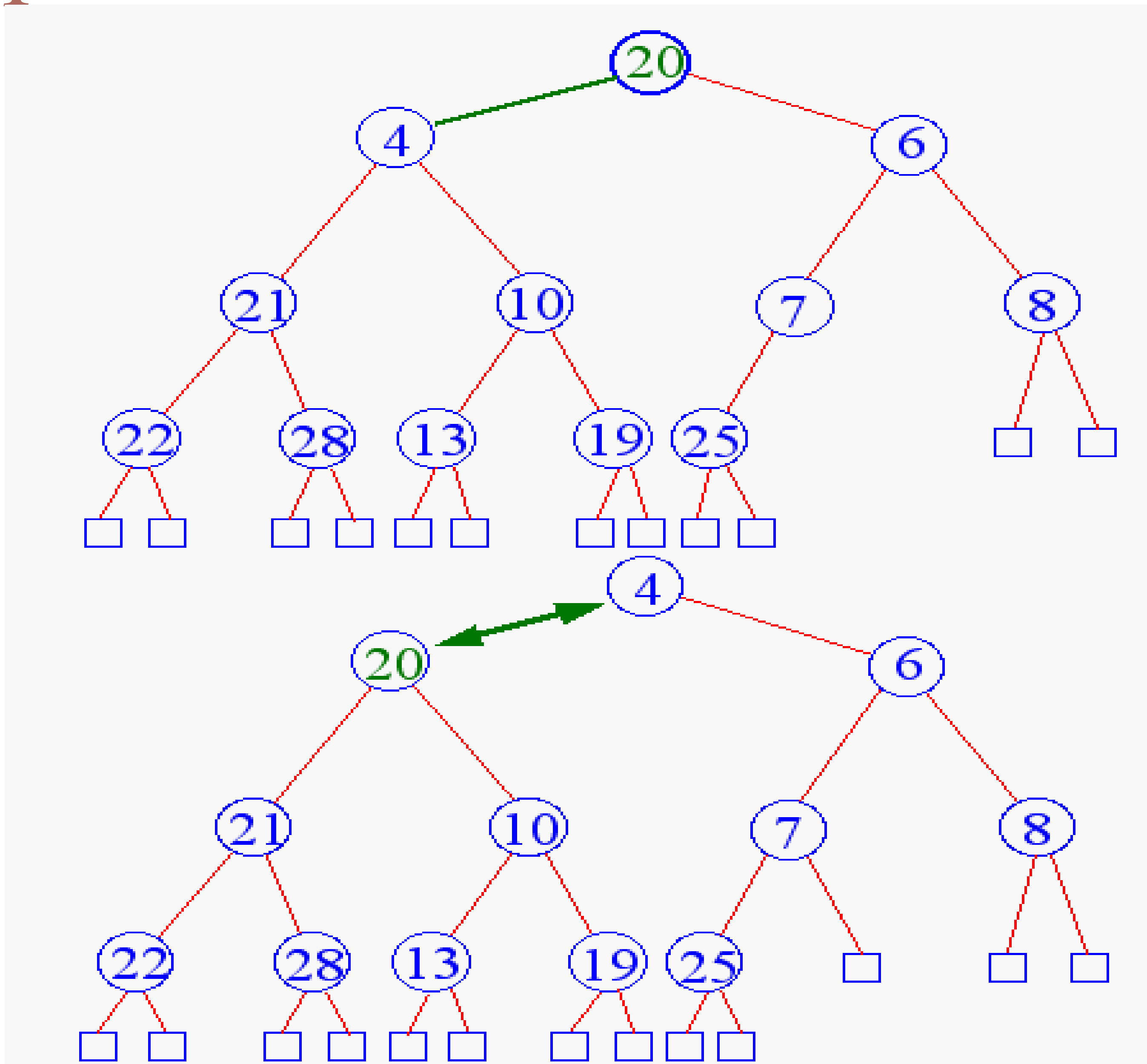
- Remove element from priority queues?

removeMin( )

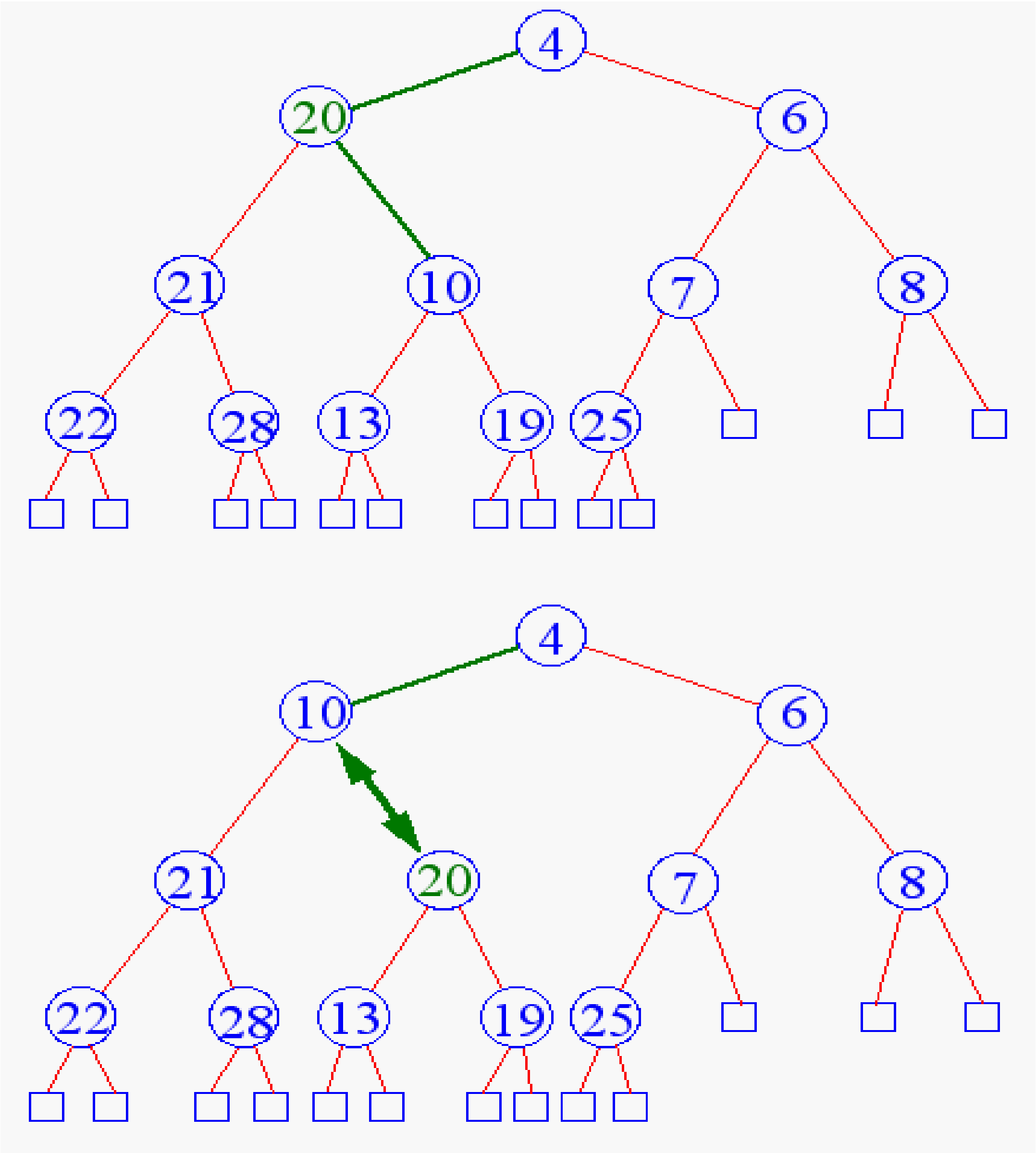


Heap Removal

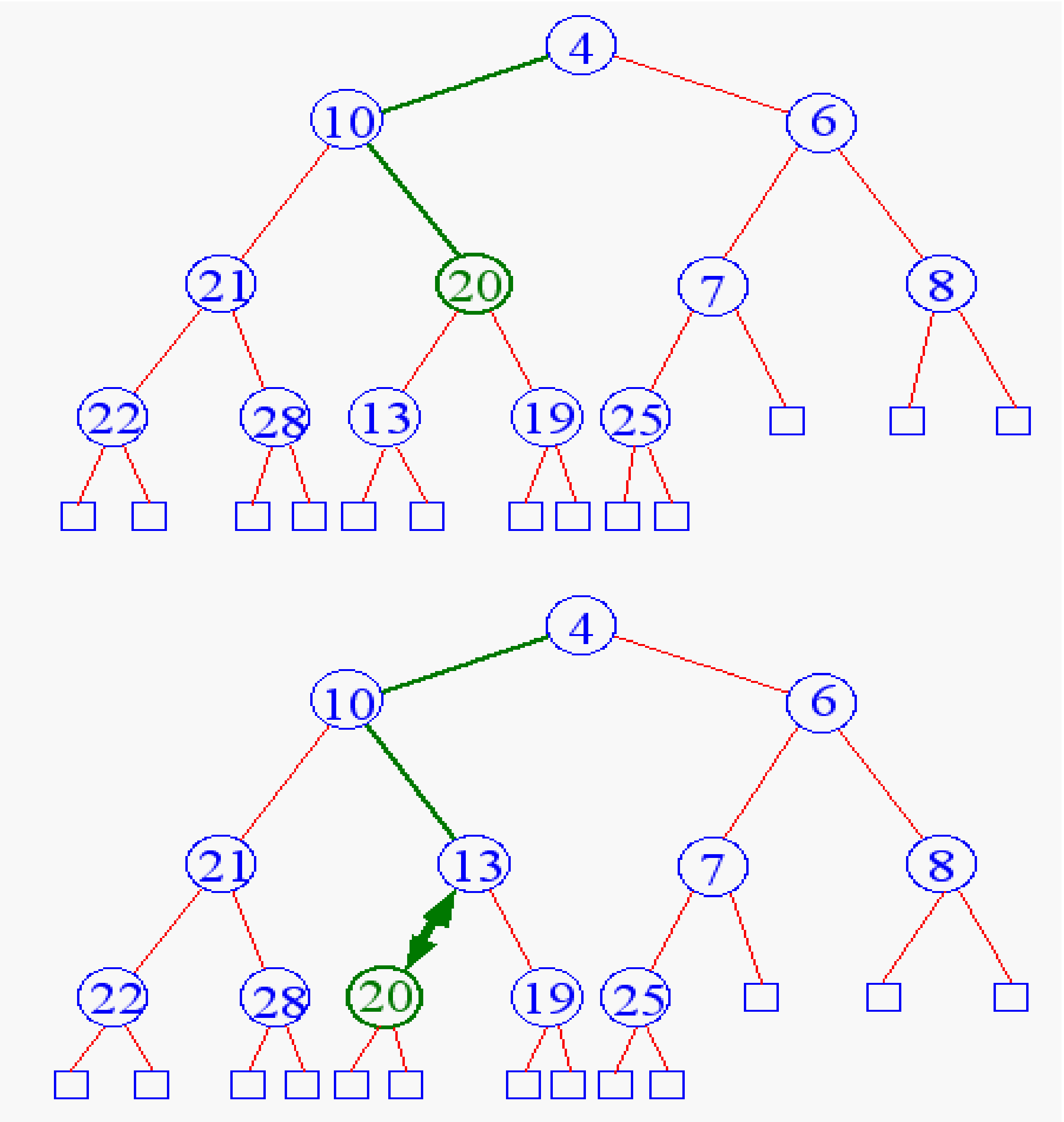
- Begin downheap



Heap Removal

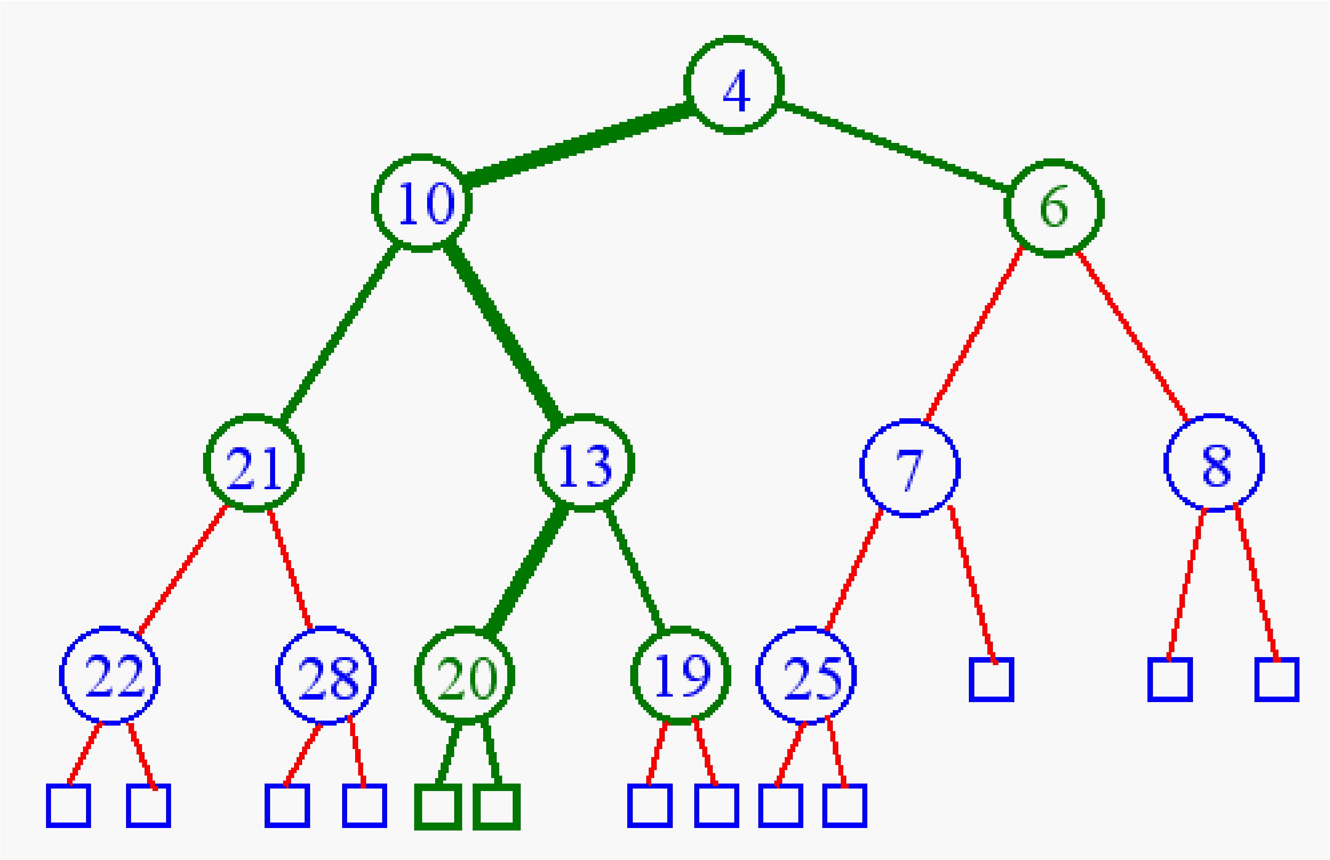


Heap Removal



Heap Removal

- Terminate downHeap when
  - reach leaf level
  - key child is greater than key parent



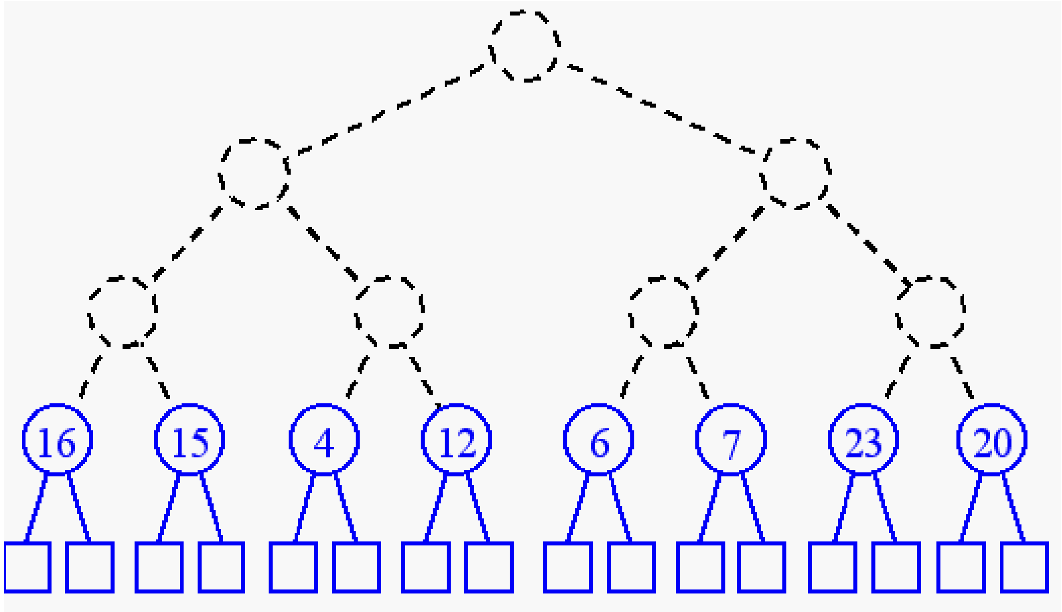
Building a Heap

- Consider building a heap using the following numbers:

  14, 9, 8, 25, 5, 11, 27, 16, 15, 4, 12, 6, 7, 23, 20

Building a Heap

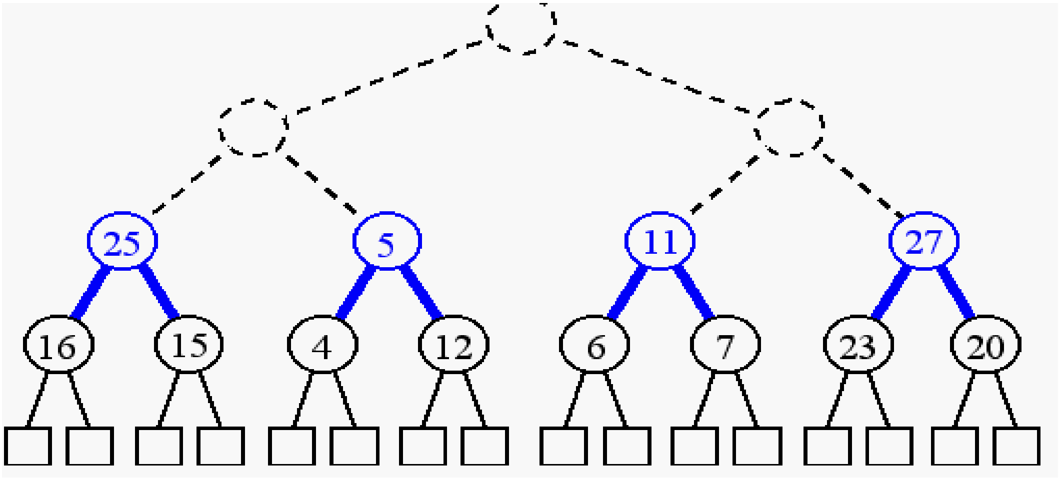
- build (n + 1)/2 trivial one-element heaps



[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
14	9	8	25	5	11	27	16	15	4	12	6	7	23	20

Building a Heap

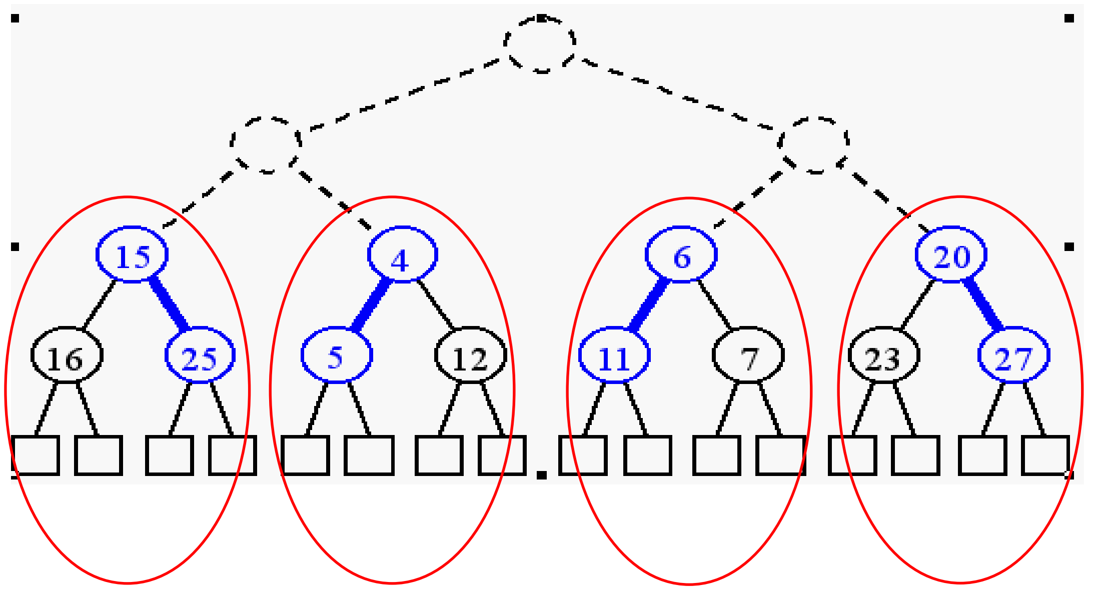
- build three-element heaps on top of them



[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
14	9	8	25	5	11	27	16	15	4	12	6	7	23	20

Building a Heap

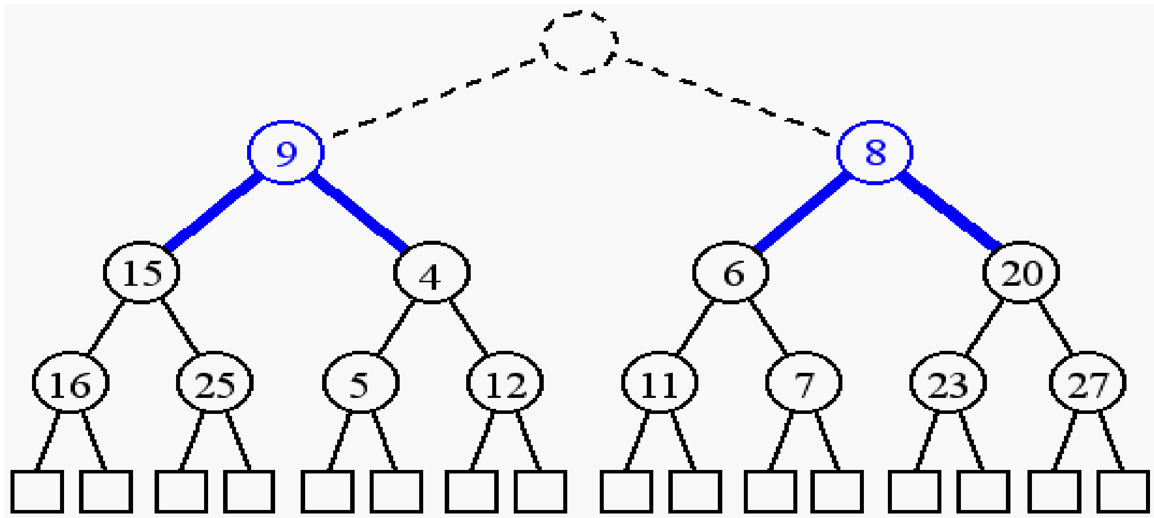
- *downheap* to preserve the order property



[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
14	9	8	15	4	6	20	16	25	5	12	11	7	23	27

Building a Heap

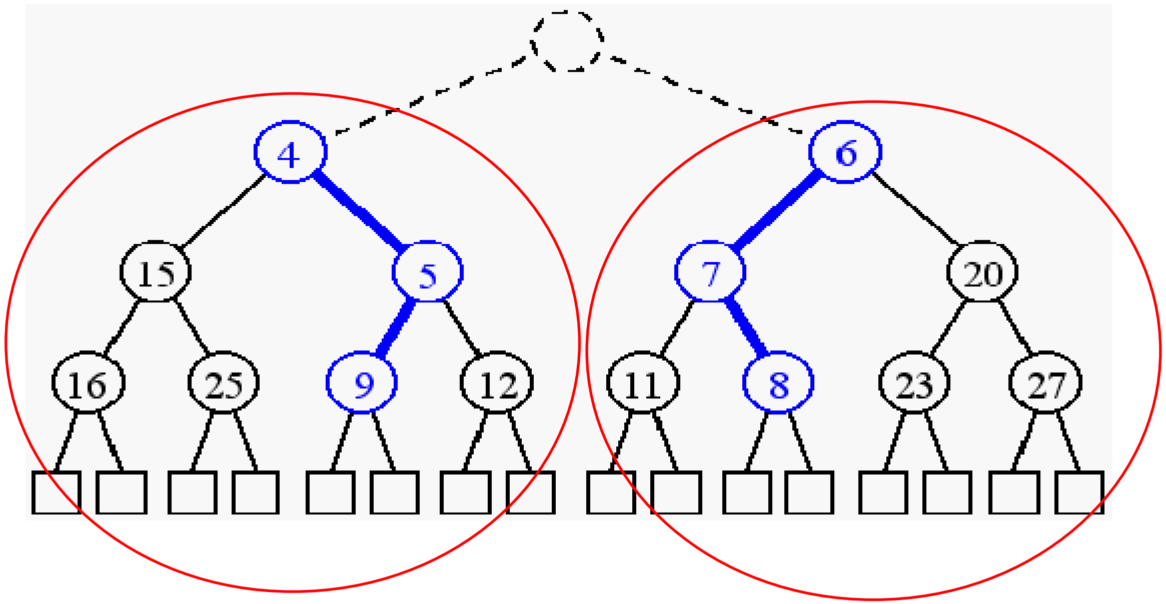
- now form seven-element heaps



[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
14	9	8	15	4	6	20	16	25	5	12	11	7	23	27

Building a Heap

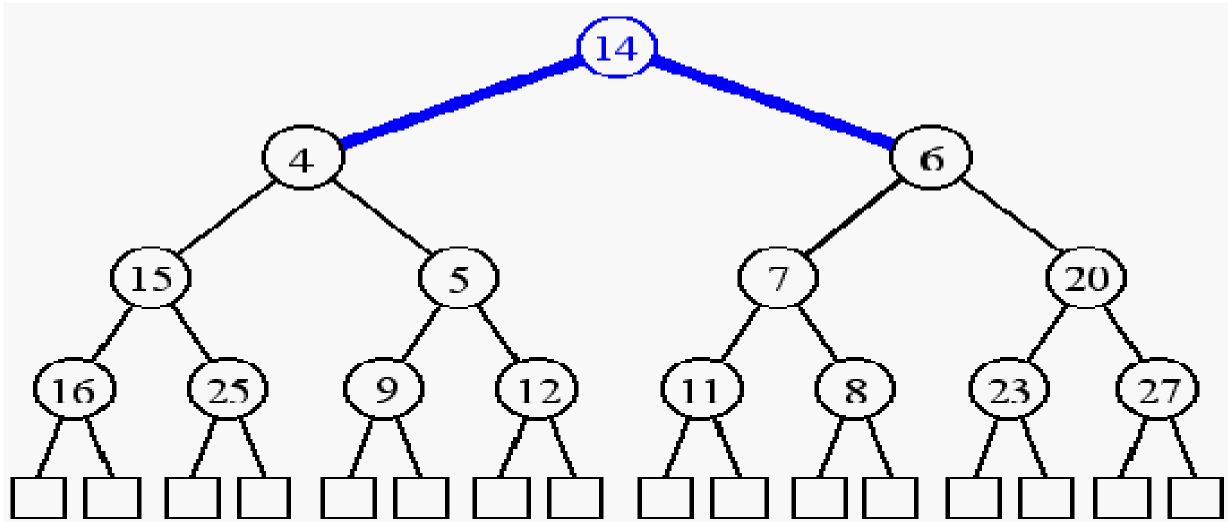
- DownHeap to preserve the order property



[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
14	4	6	15	5	7	20	16	25	9	12	11	8	23	27

Building a Heap

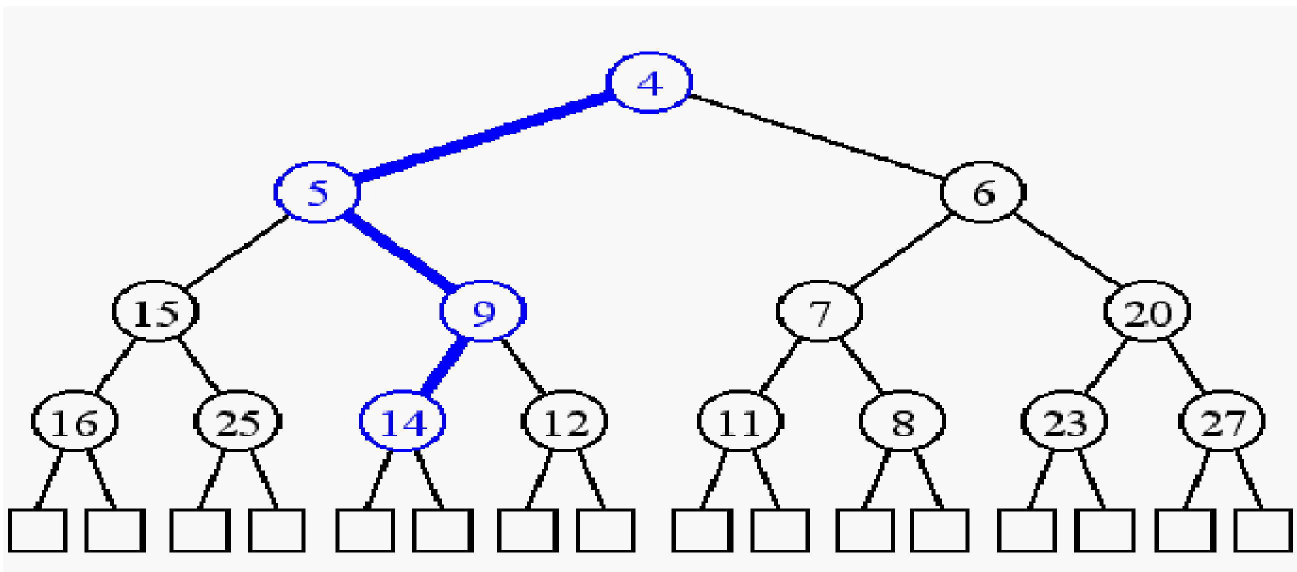
- now the last step



[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
14	4	6	15	5	7	20	16	25	9	12	11	8	23	27

Building a Heap

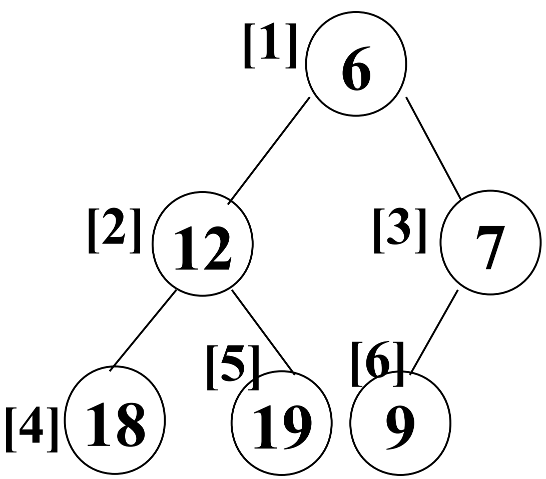
- DownHeap to preserve the order property

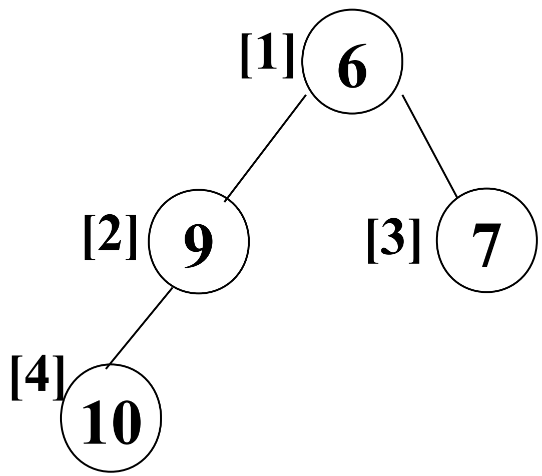


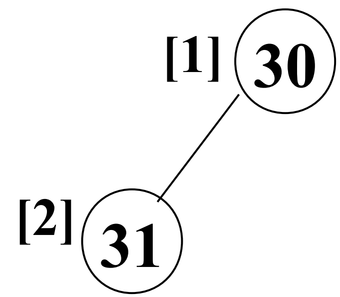
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]
4	5	6	15	9	7	20	16	25	14	12	11	8	23	27

Heap Implementation

- Using arrays
- Parent = k ; Children = 2k , 2k+1
- Why is it efficient?







[1]	[2]	[3]	[4]	[5]	[6]
6	12	7	18	19	9

[1]	[2]	[3]	[4]
6	9	7	10

[1]	[2]
30	31

More details

- Arr[1] stores the root, and Arr[k] stores the parent of Arr[2k] and Arr[2k+1], where index instarts at 1.
  - This is by default in these slides and in this class
  - For a node stored at Arr[k], its parent must be at Arr[k/2]

- Or Arr[0] stores the root, and Arr[k] stores the parent of Arr[2k+1] and Arr[2k+2], where index starts at 0.

- Consider the index of n/2 where n is the number of elements

  e.g. 9/2 = 4, 10/2 = 5

Insert into a heap

// assume heap's first position in the vector is at [1]
// last position is at [heap_size]

```
InsertHeap (heap_vector v, int heap_size, element e)
   if ( heap is not full )
      Increase heap_size by 1
      v[heap_size] ← e // append to last position
      // the following is upHeap
      i ← heap_size

      While ( i > 1 && v[ i/2 ] > v[ i ] )
         Swap v[ i ] and v[ i/2 ] //[i/2] is parent
         i ← i/2; // go to parent
```

Deletion from a heap

```
deleteHeap(heap_vector v, int heap_size)
  if heap is not empty
     v[1] ← v[heap_size]
     heap_size <-- heap_size - 1
     // the following is downHeap
     parent← 1        // parent and child are indexes
     child ← 2 X parent // left child of parent
     while child is no greater than heap_size
        if ( child is less than heap_size
             AND v[ child ] > v[ child + 1 ] )
             increase child by 1 // becomes right child
        if ( v[ parent ] < v[ child ] )
            swap v[ parent ] and v[ child ];
            parent ← child   // move down
            child ← 2 x parent // left child
```

Build a heap

- Build a heap out of a vector v of n elements (index starts at 1)
- Location $n/2$ stores the last non-leaf node.

- Algorithm of building a heap:

For index k from $n/2$ down to 1

Restore the heap property by **downHeap** for the tree whose root is v[k]

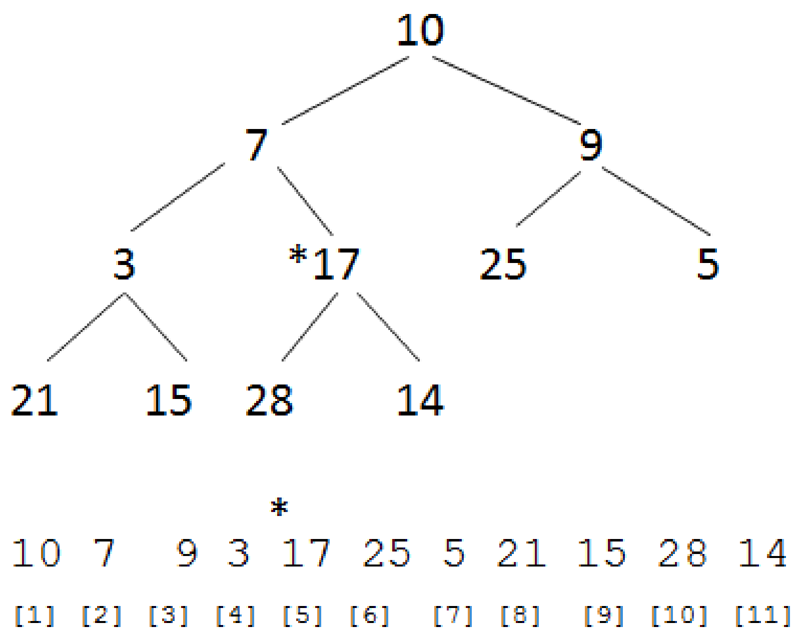
- This algorithm works for any n where a tree is not necessarily a full tree.

Build a heap

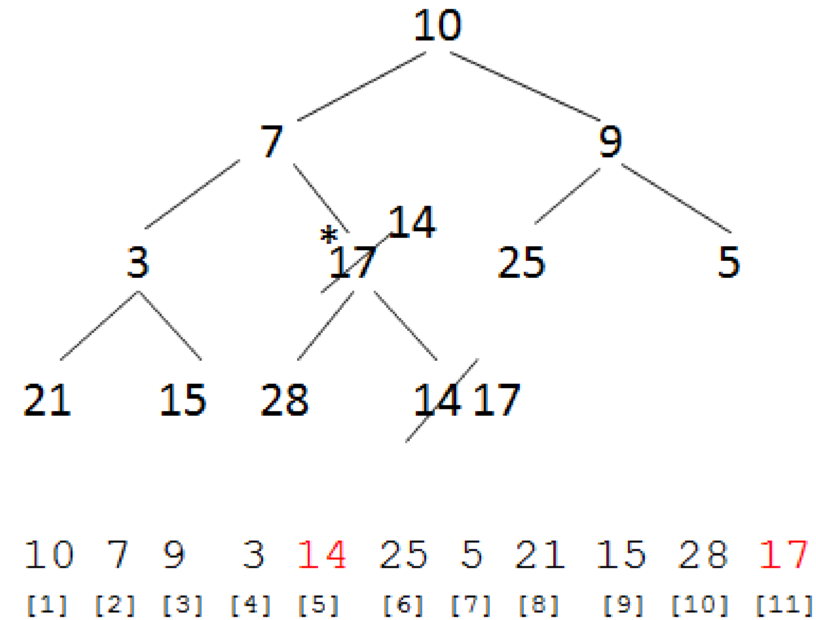
- More examples

Build a **min-heap** and a max-heap using the following sequence of data:

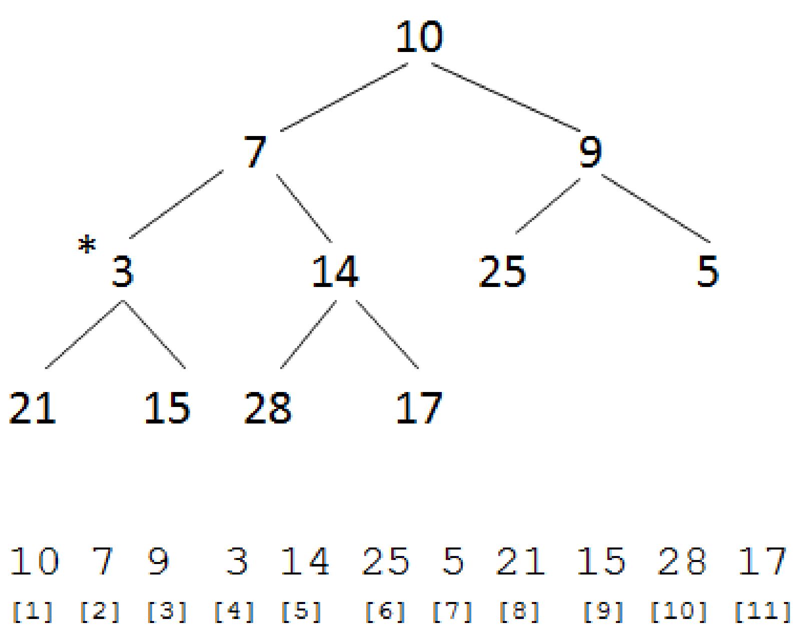
10, 7, 9, 3, 17, 25, 5, 21, 15, 28, 14


10
7
9
*17
3
25
5
21
15
28
14
10 7 9 3 *17 25 5 21 15 28 14
[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]

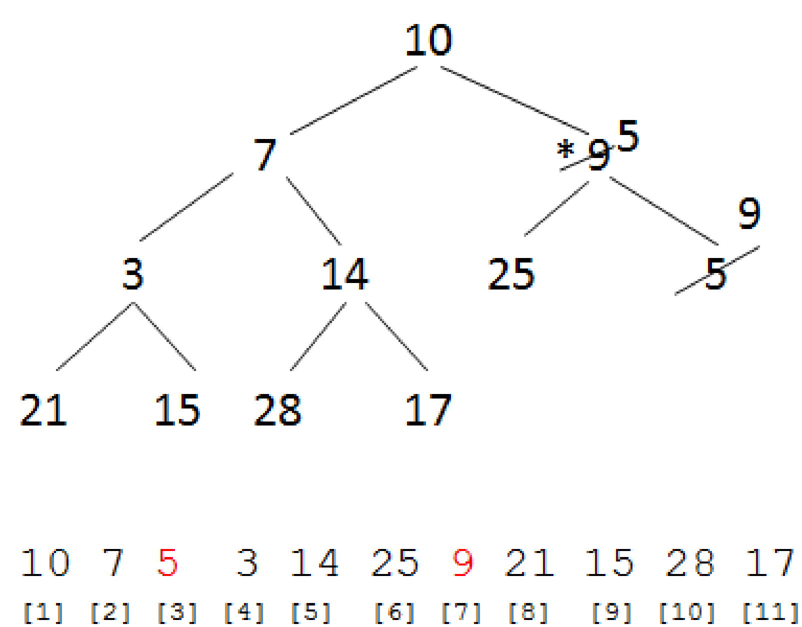
(a)


10
7
9
3
*17 14
25
5
21
15
28
14 17
10 7 9 3 14 25 5 21 15 28 17
[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]

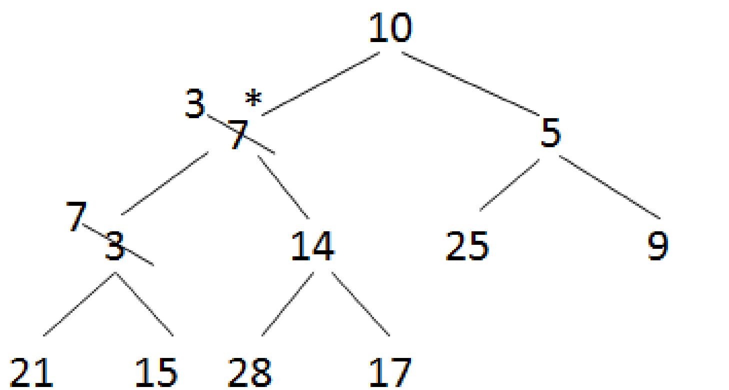
(b)


10
7
9
*3
14
25
5
21
15
28
17
10 7 9 3 14 25 5 21 15 28 17
[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]

(c)

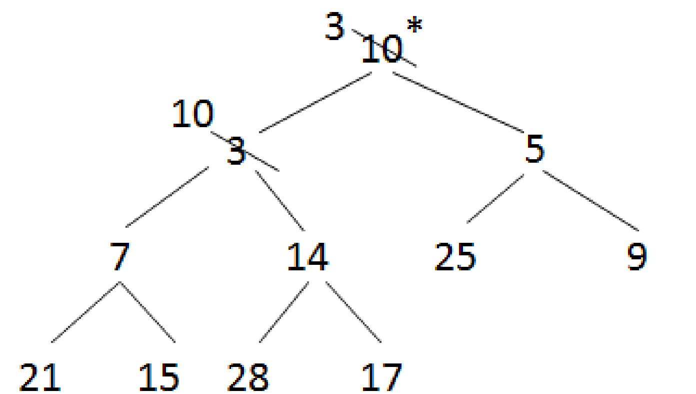

10
7
*9 5
3
14
25
5 9
21
15
28
17
10 7 5 3 14 25 9 21 15 28 17
[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11]

(d)



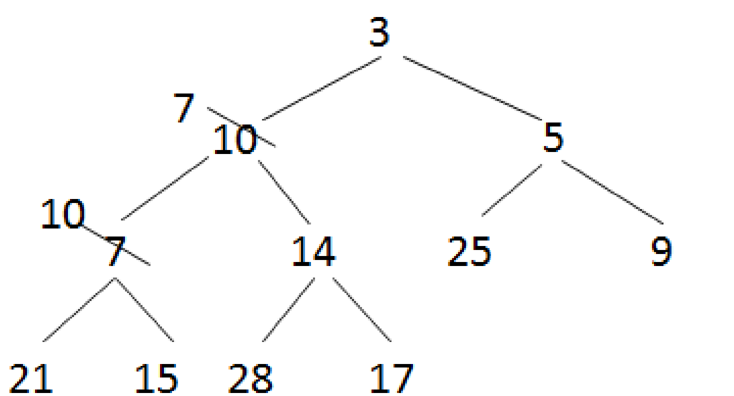
10	3	5	7	14	25	9	21	15	28	17
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

(e)



3	10	5	7	14	25	9	21	15	28	17
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

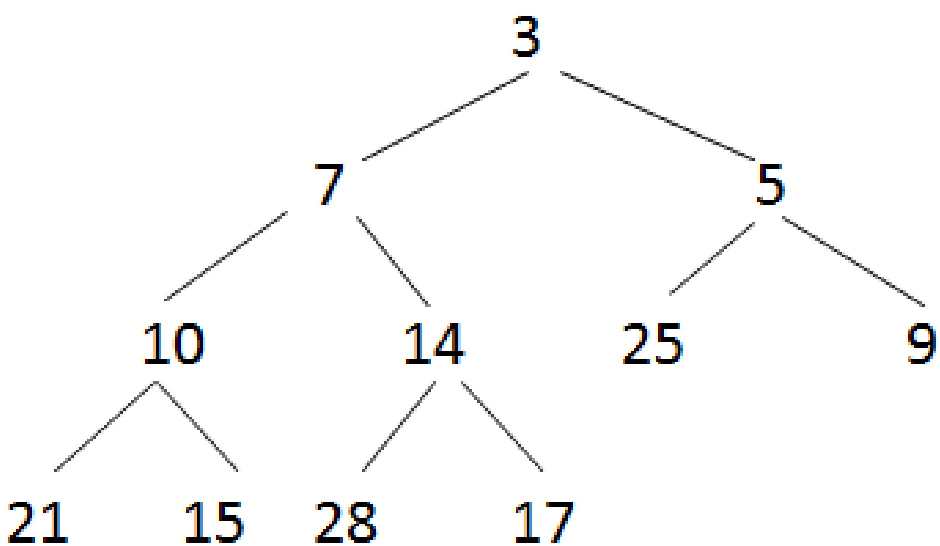
(f)



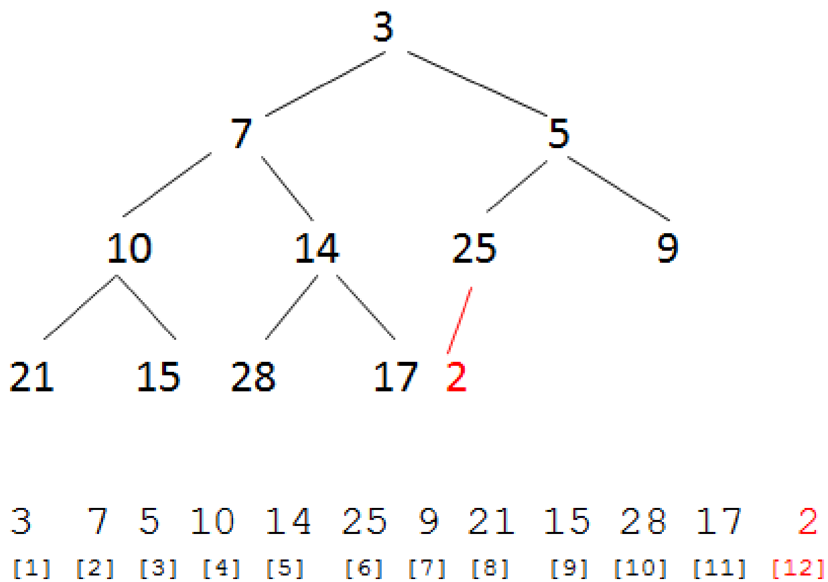
3	7	5	10	14	25	9	21	15	28	17
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

(g)

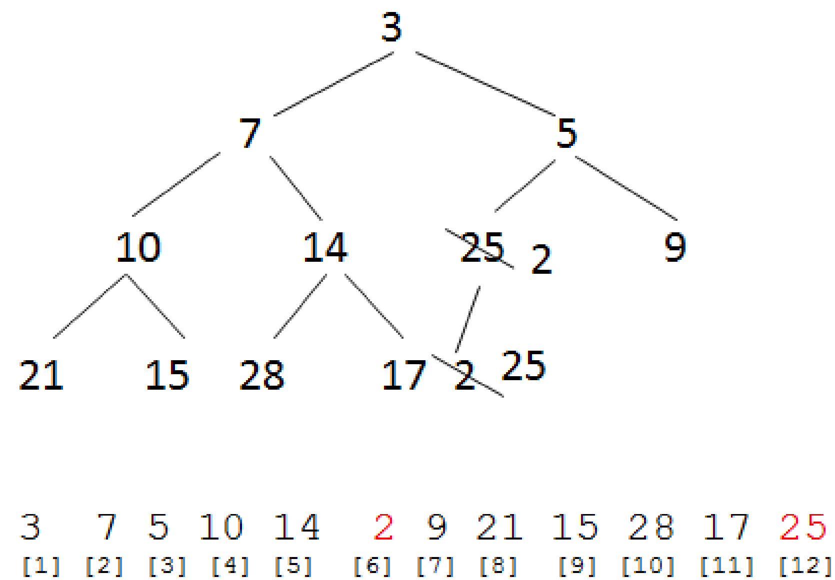
More examples: Insert 2 to current heap



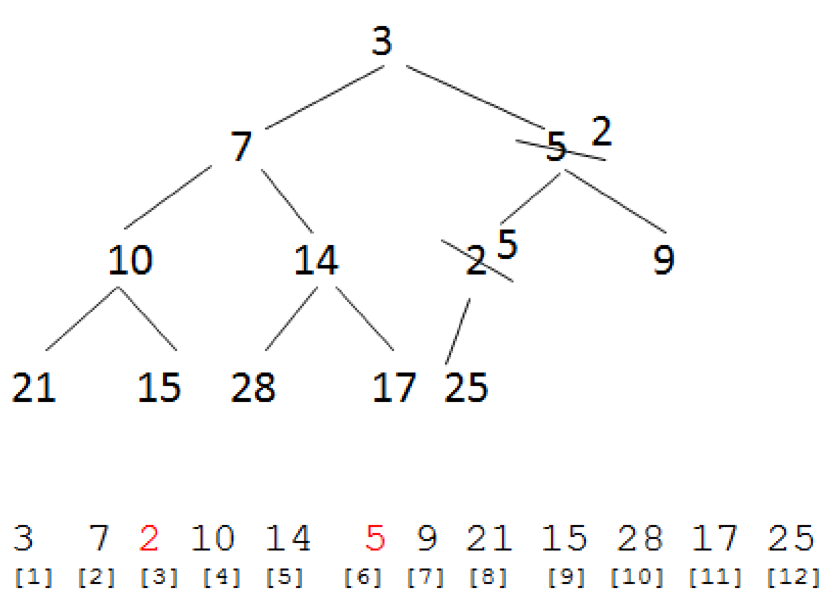
3	7	5	10	14	25	9	21	15	28	17
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]


3
7 5
10 14 25 9
21 15 28 17 2
3 7 5 10 14 25 9 21 15 28 17 2
[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12]

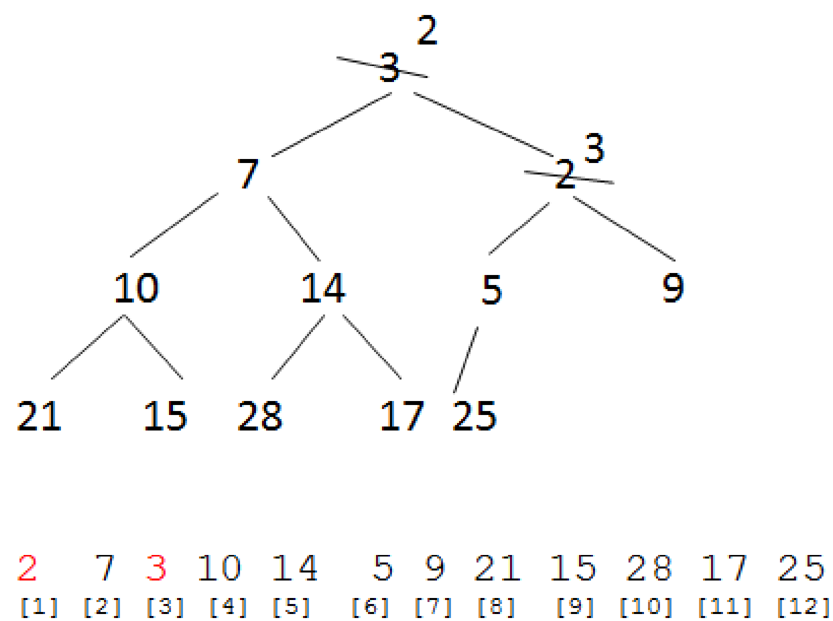
(a)


3
7 5
10 14 25 2 9
21 15 28 17 2 25
3 7 5 10 14 2 9 21 15 28 17 25
[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12]

(b)

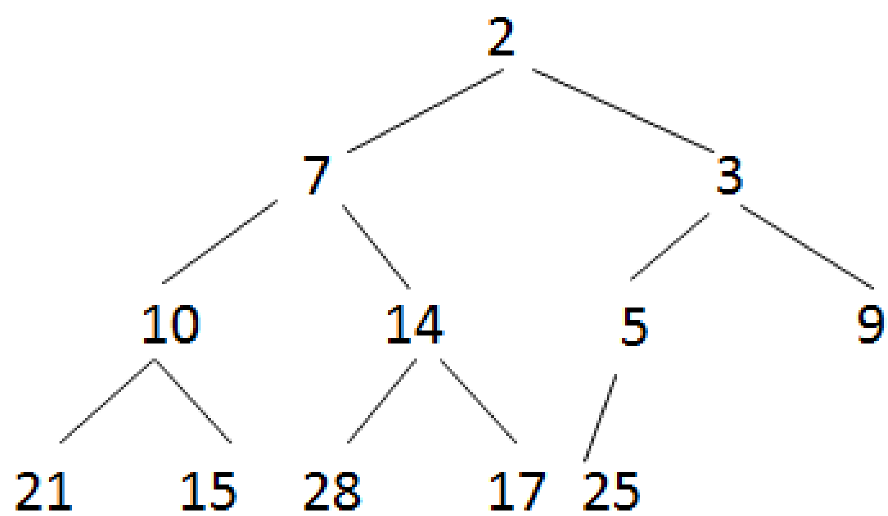

3
7 5 2
10 14 2 5 9
21 15 28 17 25
3 7 2 10 14 5 9 21 15 28 17 25
[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12]

(c)

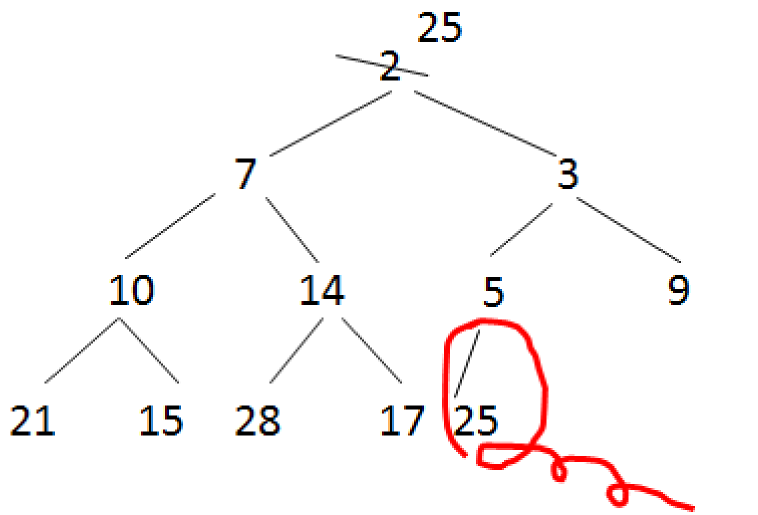

2
3
7 2 3
10 14 5 9
21 15 28 17 25
2 7 3 10 14 5 9 21 15 28 17 25
[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12]

(d)

More examples: remove/delete/extract min

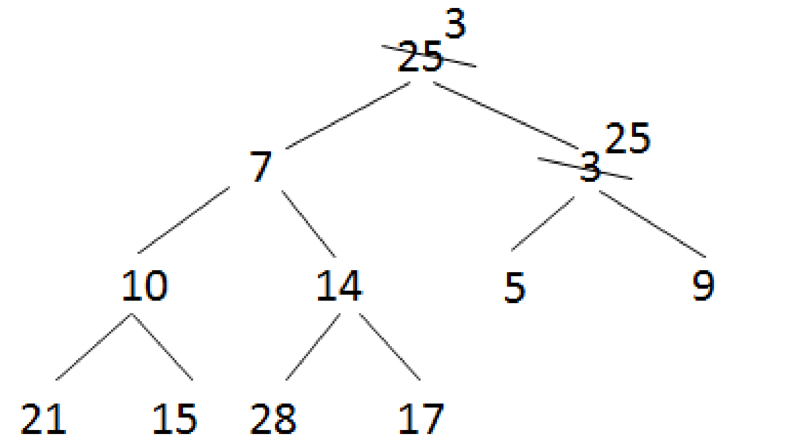


2	7	3	10	14	5	9	21	15	28	17	25
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]



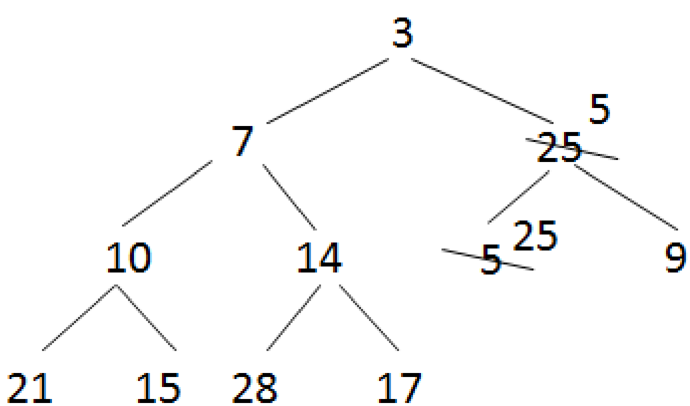
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
25	7	3	10	14	5	9	21	15	28	17

(a)



[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
3	7	25	10	14	5	9	21	15	28	17

(b)



[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]
3	7	5	10	14	25	9	21	15	28	17

(c)

HeapSort - sort a sequence of data

- Step 1: Build a max-heap
- Step 2: repetitive call to deleteHeap( )

```
while n >=  2
        swap v [ n ] and v[1]   // now the current max is saved
                                // to the end of current heap
        n <-- n-1  // reduce the heap size by 1
        downHeap (v, n)
```

- Running time? -- consider a full tree

$$n = 2^k-1 \implies k = \lceil \log_2(n+1) \rceil$$

height of heap: $O(\log_2 n)$

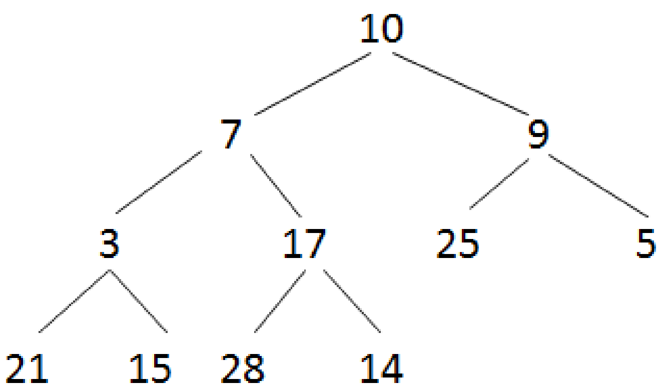
HeapSort

- More examples

Sort the following sequence of data:

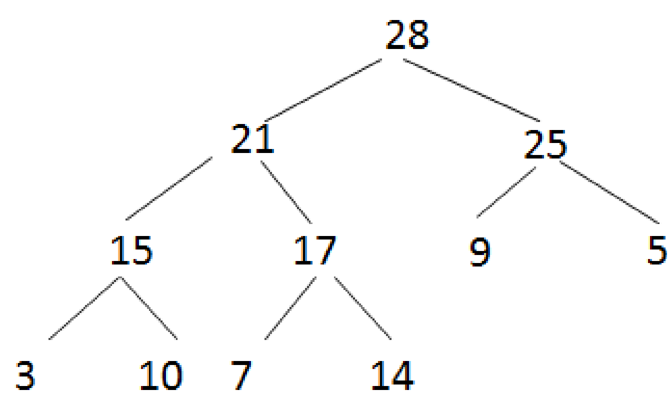
```
10 7  9 3 17 25 5 21 15 28 14
```

First major step: build a max-heap



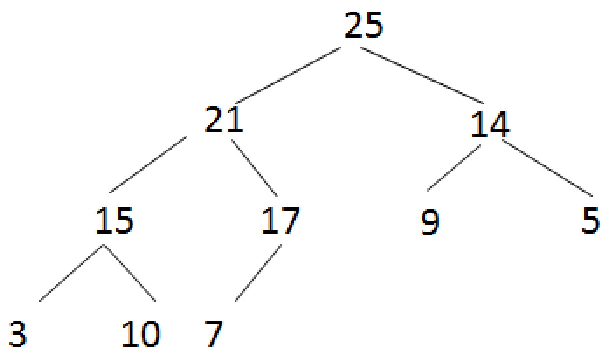
10	7	9	3	17	25	5	21	15	28	14
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

Many swaps

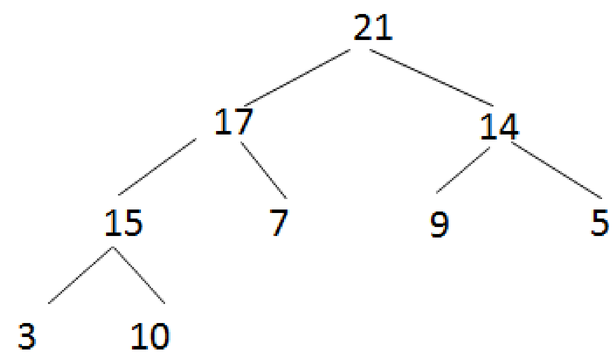


28	21	25	15	17	9	5	3	10	7	14
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

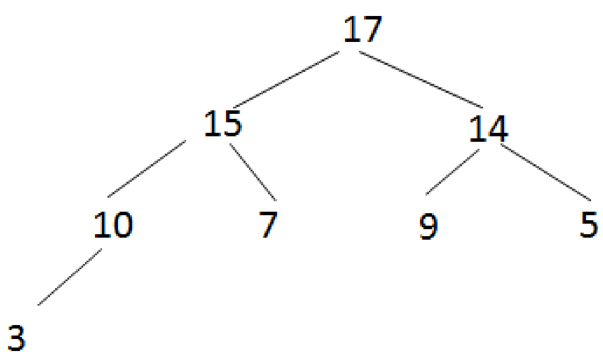
2nd major step: a series of deletion (max) operations



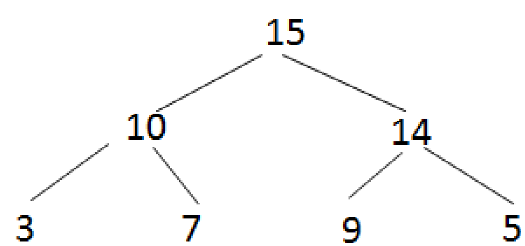
25	21	14	15	17	9	5	3	10	7	28
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]



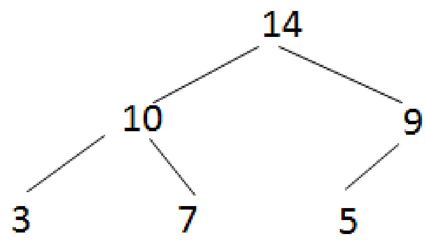
21	17	14	15	7	9	5	3	10	25	28
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]



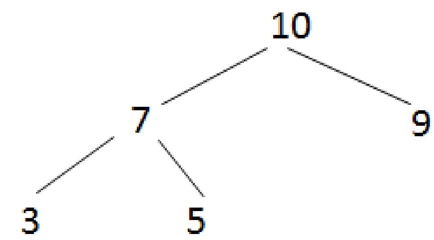
17	15	14	10	7	9	5	3	21	25	28
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]



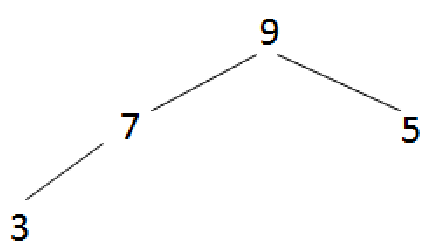
15	10	14	3	7	9	5	17	21	25	28
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]



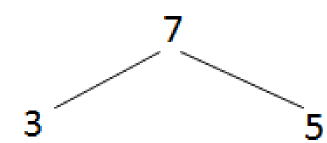
14	10	9	3	7	5	15	17	21	25	28
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]



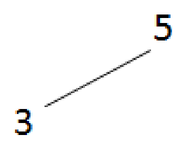
10	7	9	3	5	14	15	17	21	25	28
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]



9	7	5	3	10	14	15	17	21	25	28
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]



7	3	5	9	10	14	15	17	21	25	28
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]



5	3	7	9	10	14	15	17	21	25	28
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]

3	5	7	9	10	14	15	17	21	25	28
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]