

Algorithms

Sorting

### Overview

- Sorting in Simple Way
- Lower Bound on Comparison Sorting
- Heapsort
- Quicksort
- Sorting in Linear Time

**Selection Sort**

- Repeatedly extract an entry with minimum value from the set and append it to the sorted sequence.

15	45	7	13	5	42	2	99	8	32
----	----	---	----	---	----	---	----	---	----

**Selection Sort**

- Repeatedly extract an entry with minimum value from the set and append it to the sorted sequence.

15	45	7	13	5	42	2	99	8	32
----	----	---	----	---	----	---	----	---	----

$$T(n) = \sum_{i=1}^n i = \frac{(n+1)n}{2}$$

### Insertion Sort

- Build up the sorted sequence incrementally, by inserting one entry at a time in an appropriate place.

15	45	7	13	5	42	2	99	8	32
----	----	---	----	---	----	---	----	---	----

**Insertion Sort**

- Build up the sorted sequence incrementally, by inserting one entry at a time in an appropriate place.

15	45	7	13	5	42	2	99	8	32
----	----	---	----	---	----	---	----	---	----

$$T(n) = \sum_{i=1}^n i = \frac{(n+1)n}{2}$$

**Bubble Sort**

- Bring the smallest element to the front of the list, bring the second smallest element to the second position, etc.

15	45	7	13	5	42	2	99	8	32
----	----	---	----	---	----	---	----	---	----

**Bubble Sort**

- Bring the smallest element to the front of the list, bring the second smallest element to the second position, etc.

15	45	7	13	5	42	2	99	8	32
----	----	---	----	---	----	---	----	---	----

$$T(n) = \sum_{i=1}^n (i - 1) = \frac{n(n - 1)}{2}$$

**Lower Bound**

- Any comparison-based sorting algorithm needs  $\Omega(n \log n)$  time.
- Binary tree of height  $h$  has at most  $2^h$  leaves (by induction).
- Decision tree in any comparison sorting must satisfy:  $2^h \geq n!$

$$\begin{aligned} h &\geq \log n! \\ &> \log \left(\frac{n}{e}\right)^n \\ &= n \log n - n \log e \\ &= \Omega(n \log n) \end{aligned}$$

- using Stirling's approximation (P. 55)

**Merge Sort**

- **Divide** the set into two equally large subsets.
- **Sort** each subset recursively using MERGE sort.
- **Merge** the two sorted subsets.

15	45	7	13	5	42	2	99	8	32
----	----	---	----	---	----	---	----	---	----

### Merge Sort

- **Divide** the set into two equally large subsets.
- **Sort** each subset recursively using MERGE sort.
- **Merge** the two sorted subsets.

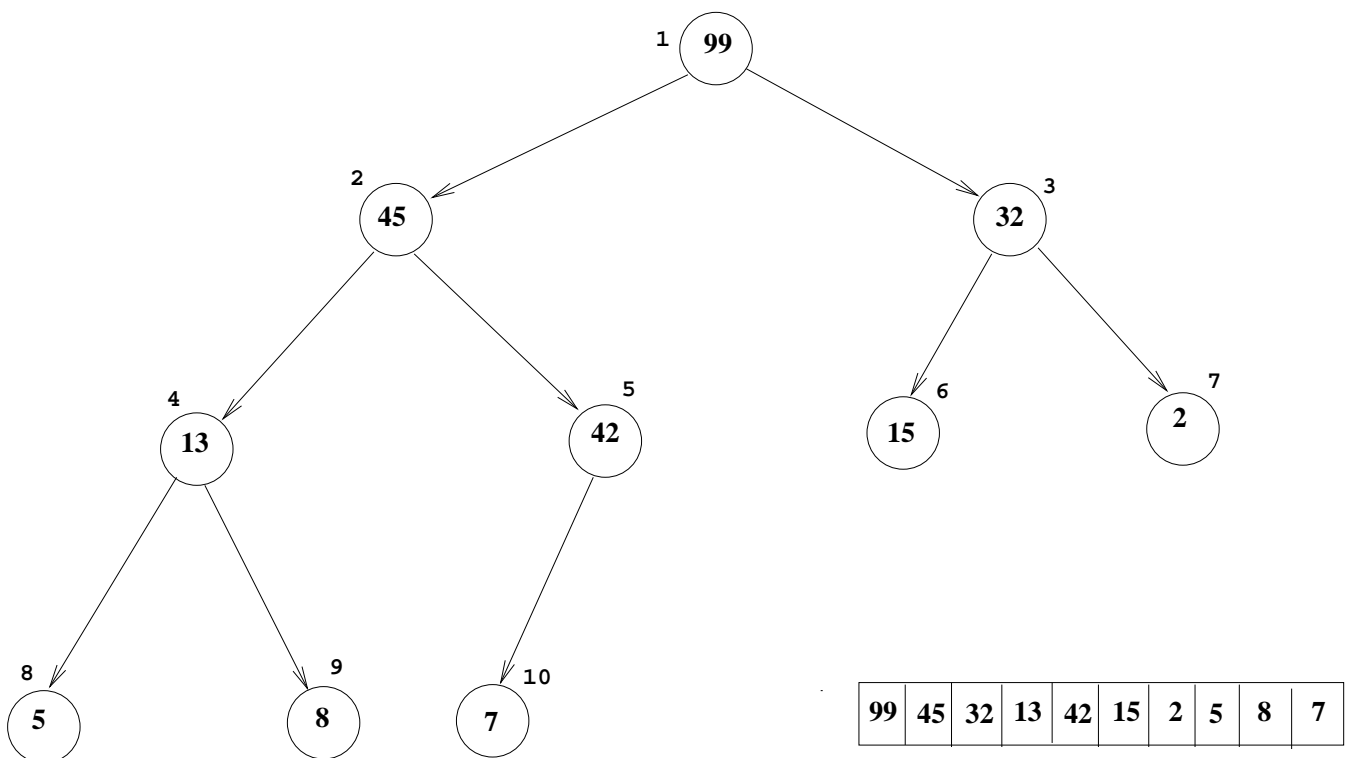
15	45	7	13	5	42	2	99	8	32
----	----	---	----	---	----	---	----	---	----

$$T(n) = \begin{cases} \Theta(1) & \text{if } n = 1 \\ 2T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

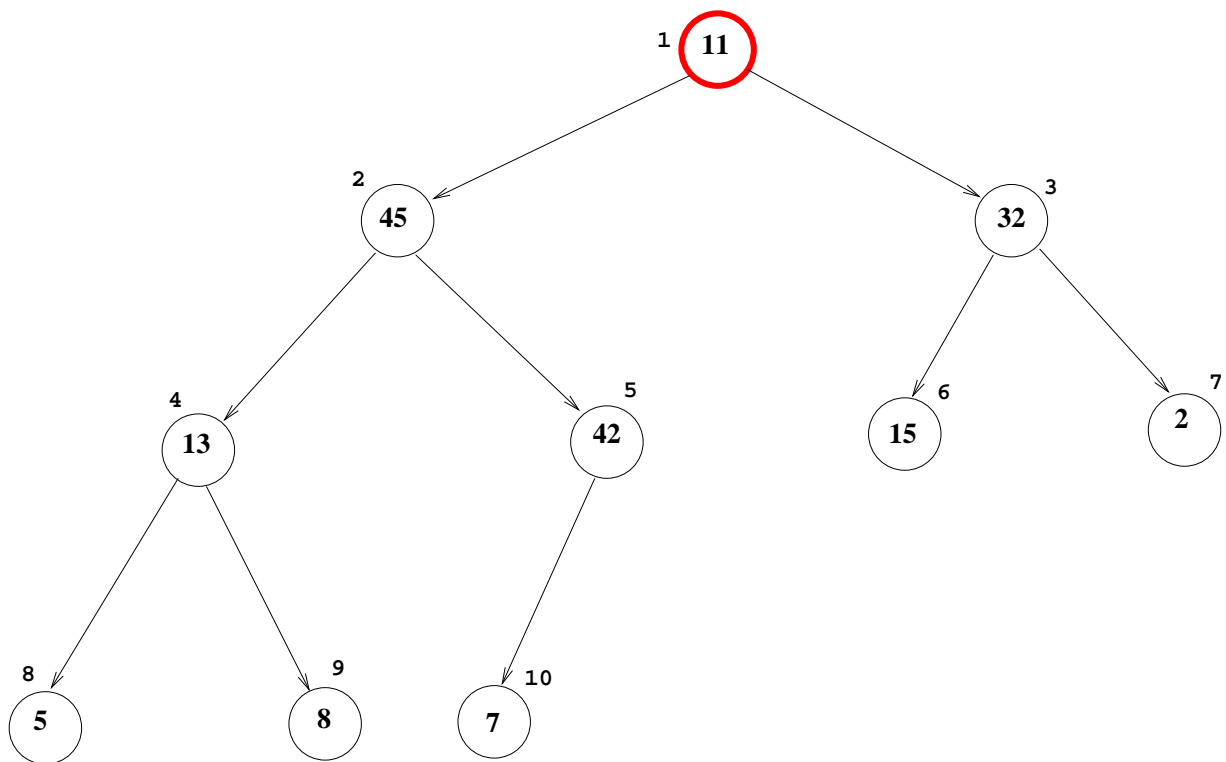
- $T(n) = O(n \log n)$  (see p. 63)

## Binary Heaps

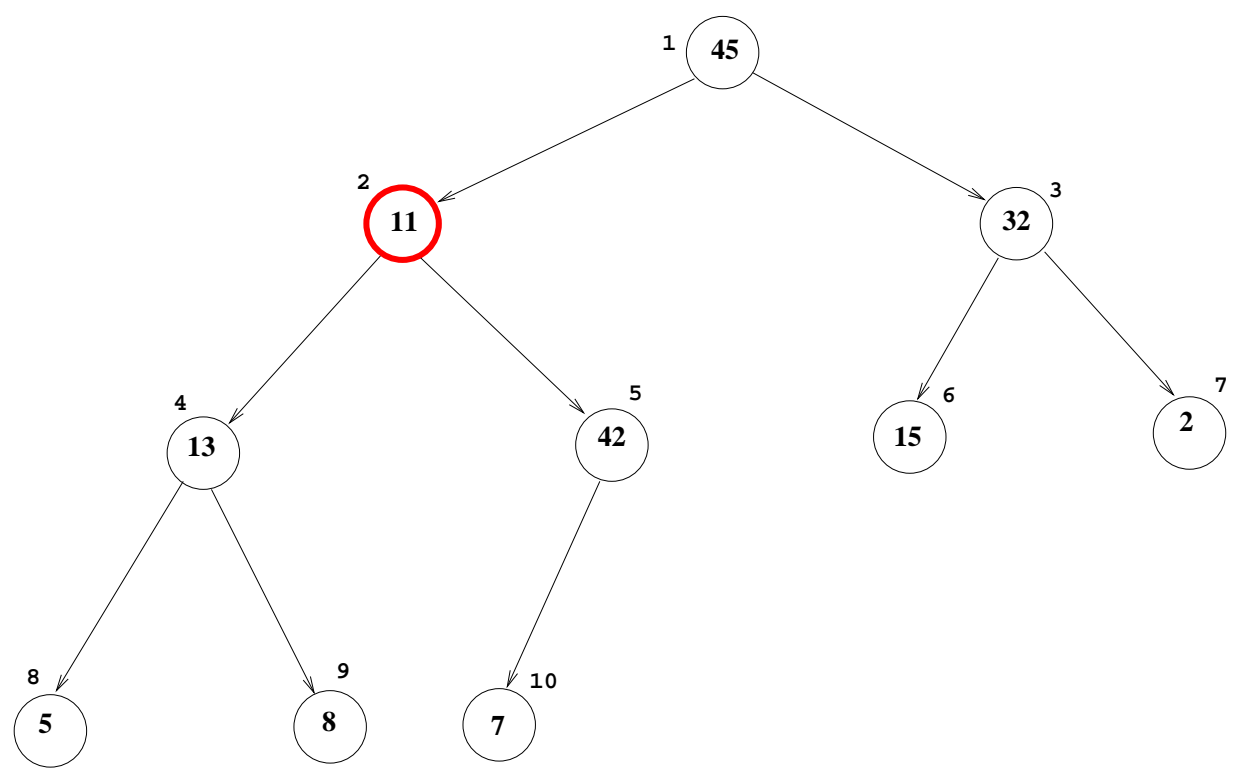
- Can be represented as simple arrays.
- Children of the node stored in entry  $i$  can be found in entries  $2i$  and  $2i + 1$ .



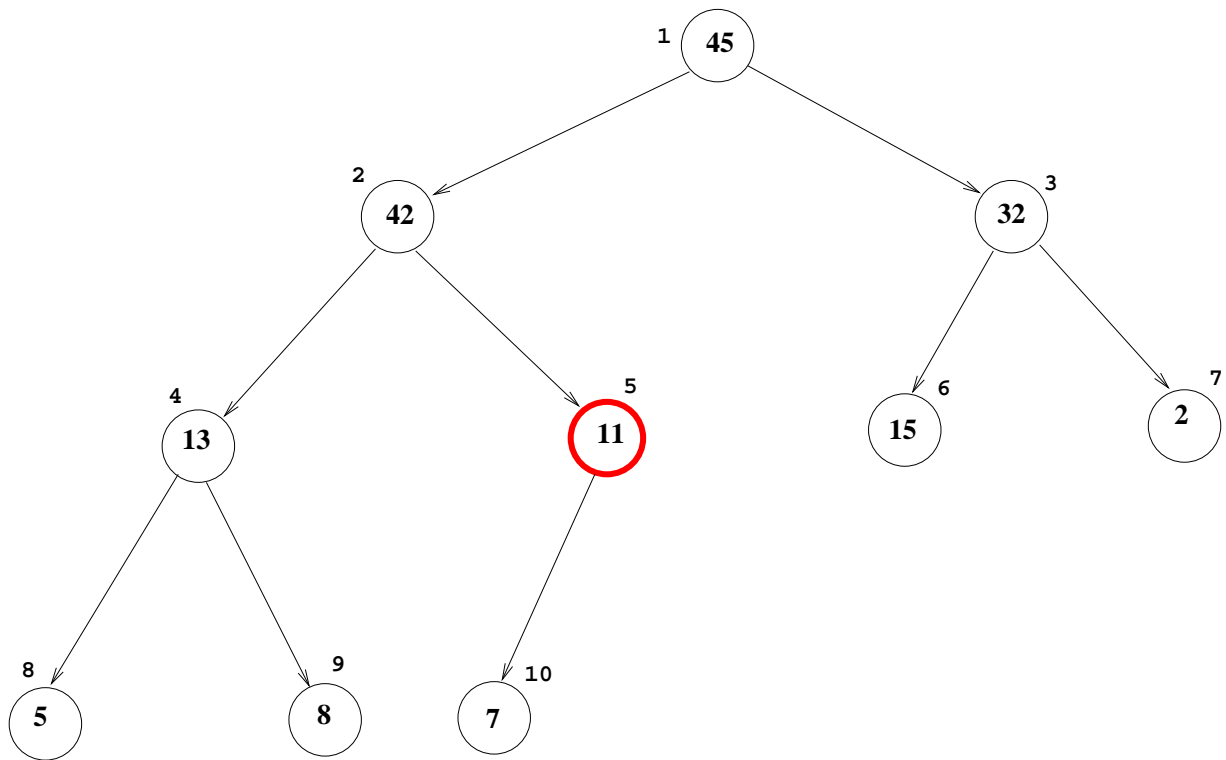
### Heapify



Heapify

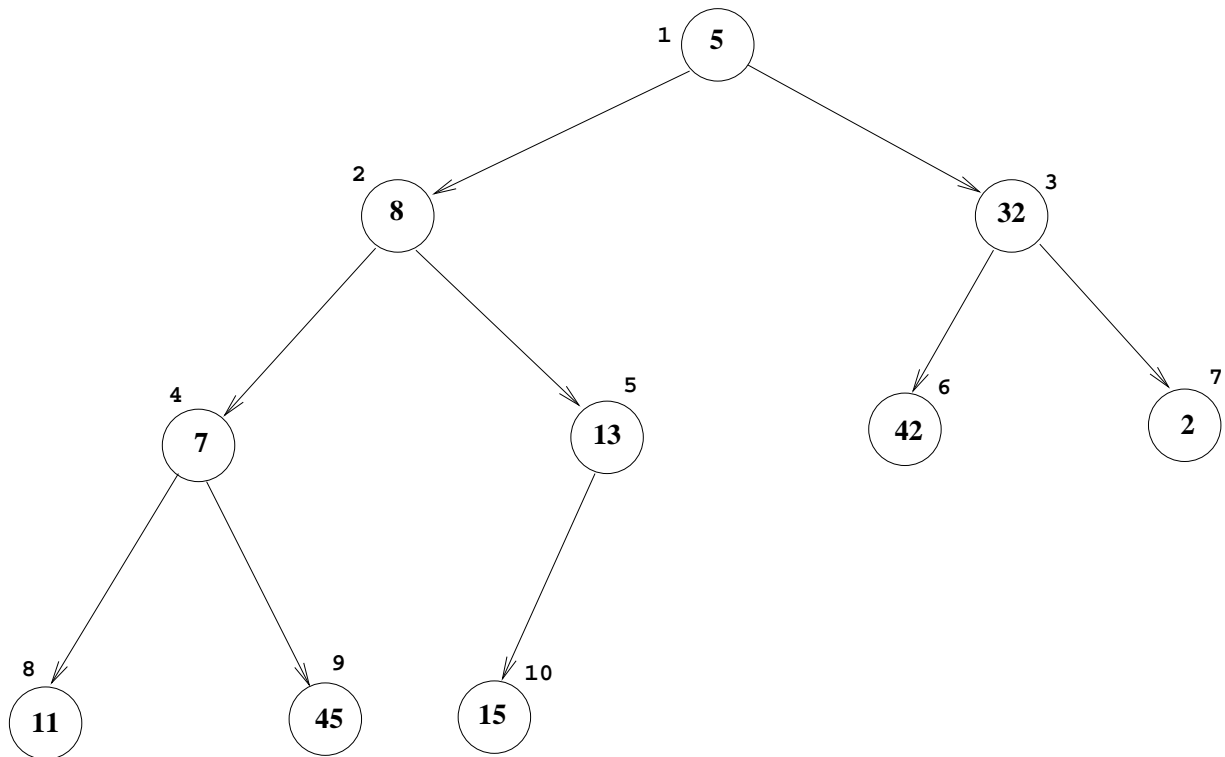


Heapify



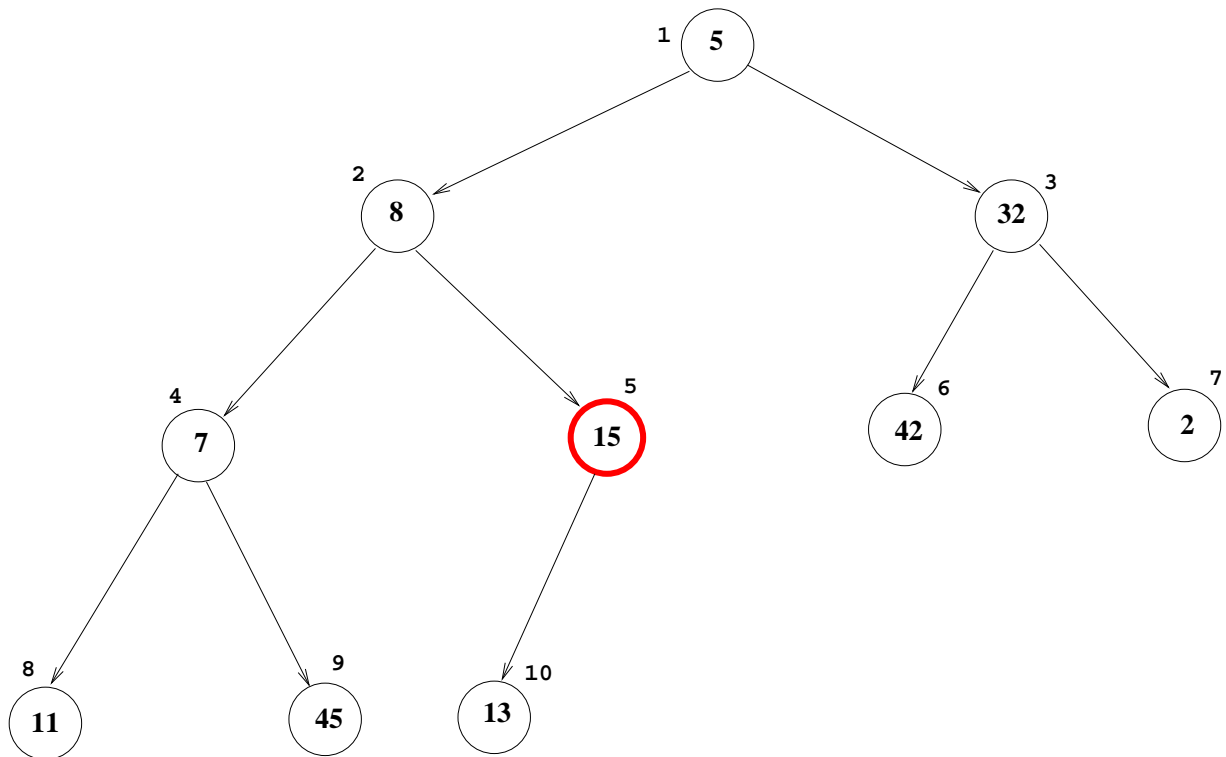
### Building a Heap

- A heap can be built by applying `HEAPIFY` to the nodes with children, starting with the one stored in the highest entry.



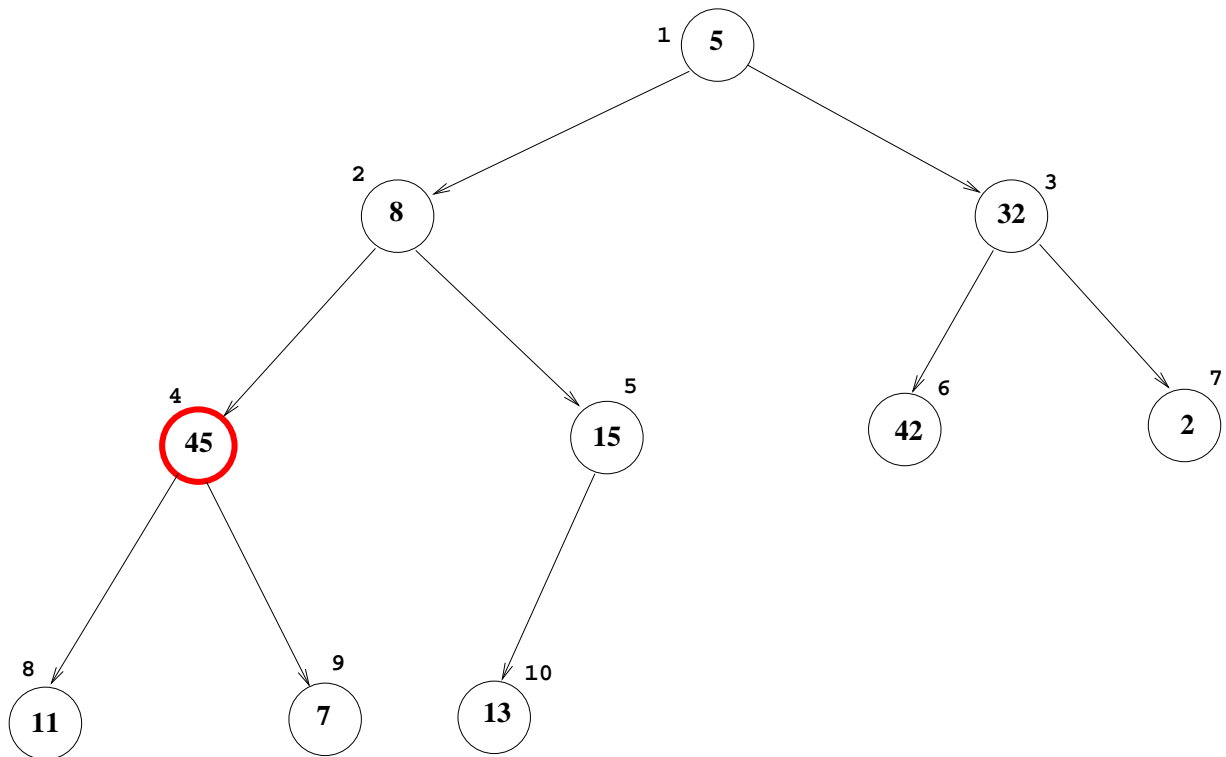
### Building a Heap

- A heap can be built by applying `HEAPIFY` to the nodes with children, starting with the one stored in the highest entry.



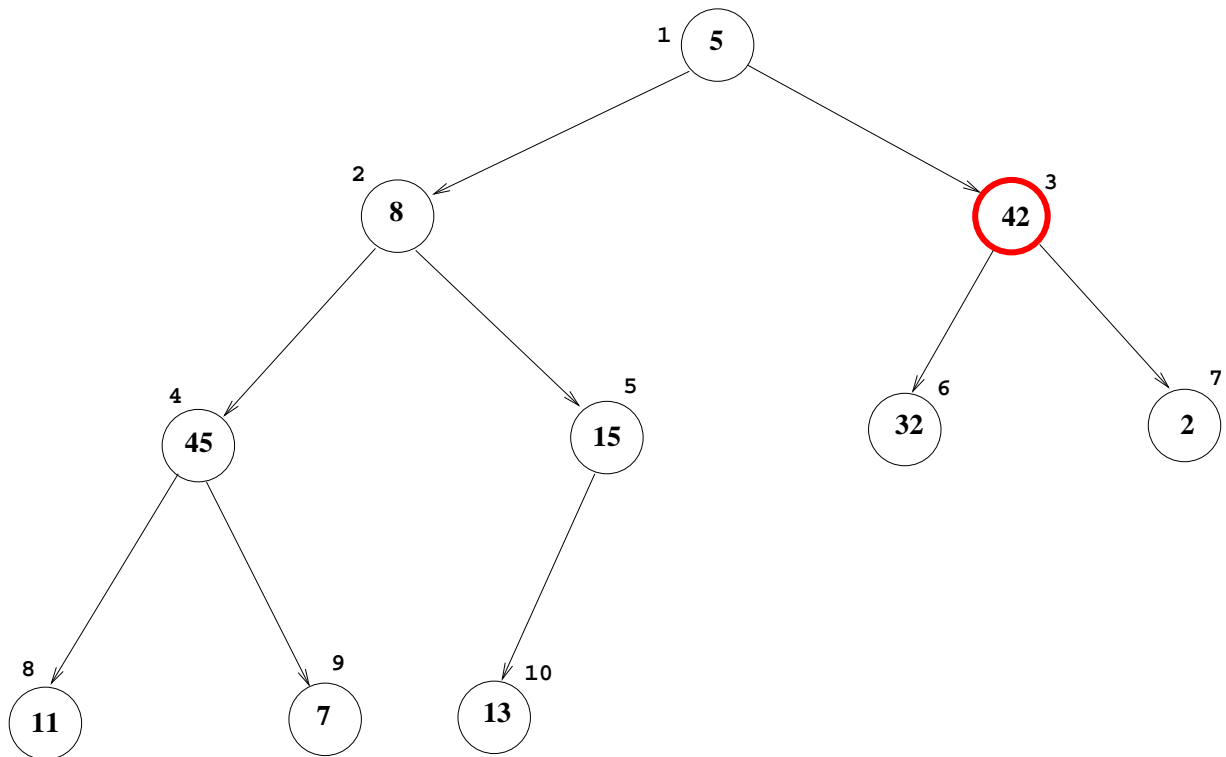
### Building a Heap

- A heap can be built by applying `HEAPIFY` to the nodes with children, starting with the one stored in the highest entry.



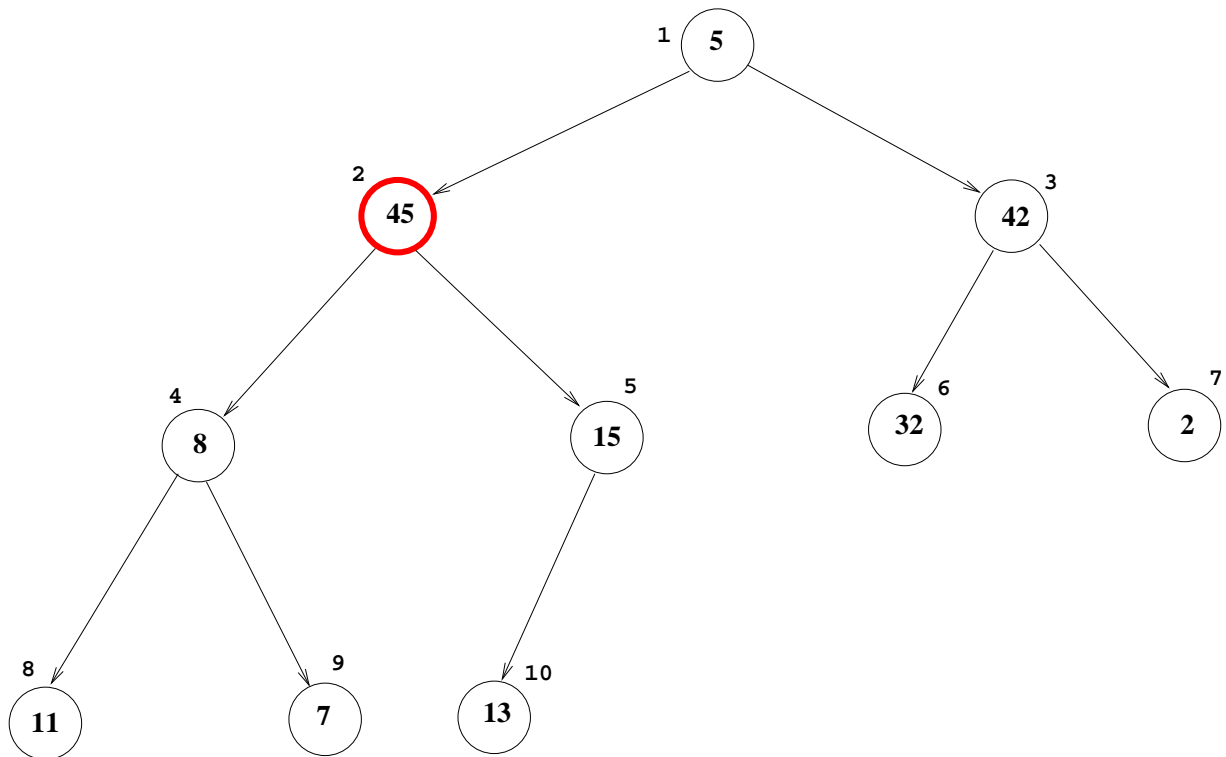
### Building a Heap

- A heap can be built by applying `HEAPIFY` to the nodes with children, starting with the one stored in the highest entry.



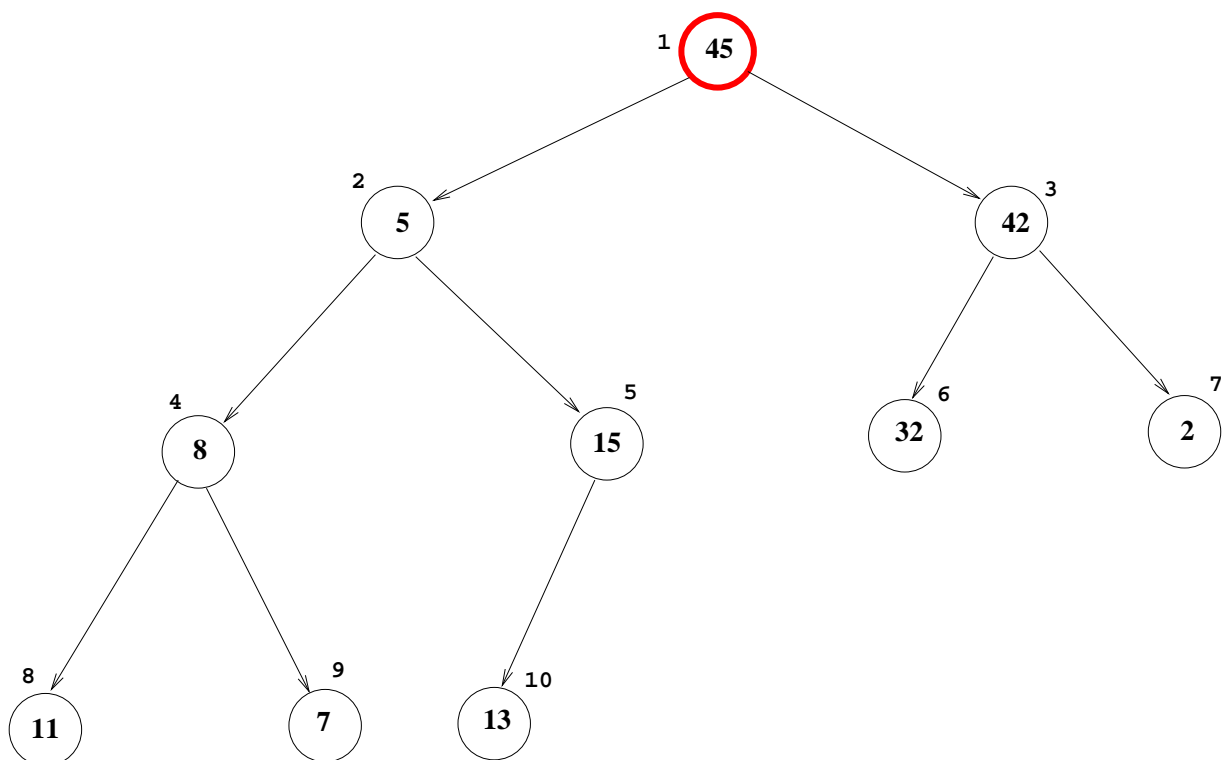
### Building a Heap

- A heap can be built by applying `HEAPIFY` to the nodes with children, starting with the one stored in the highest entry.



### Building a Heap

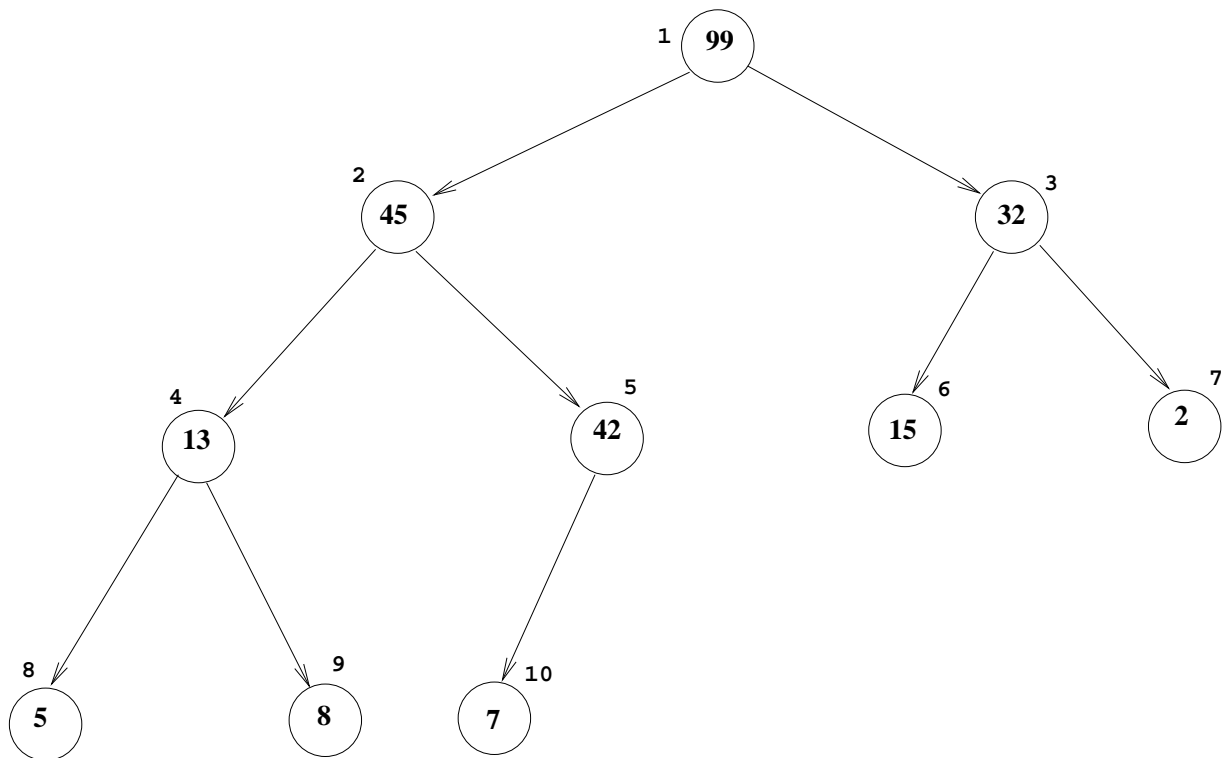
- A heap can be built by applying `HEAPIFY` to the nodes with children, starting with the one stored in the highest entry.



- Building a heap requires  $O(n \log n)$
- This analysis is not tight!

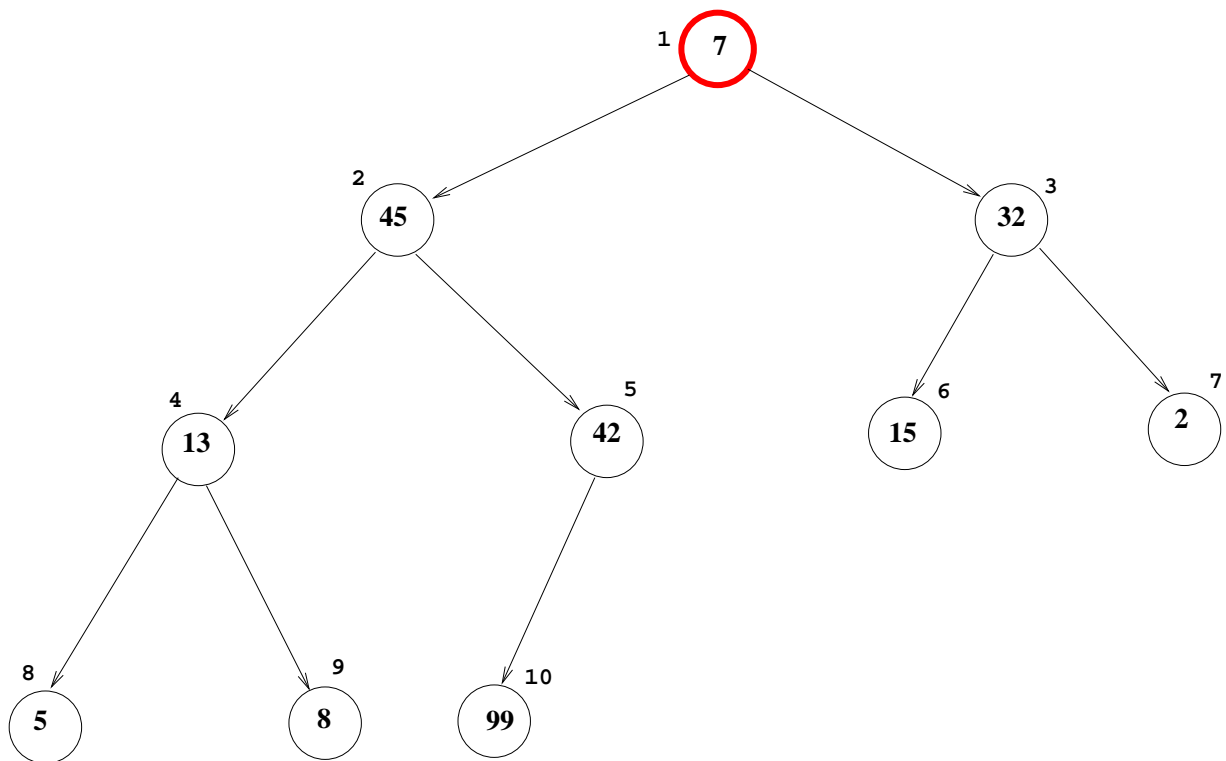
## Heapsort

- Swap the root and the node in the last entry.
- Remove last entry.
- Heapify the root.



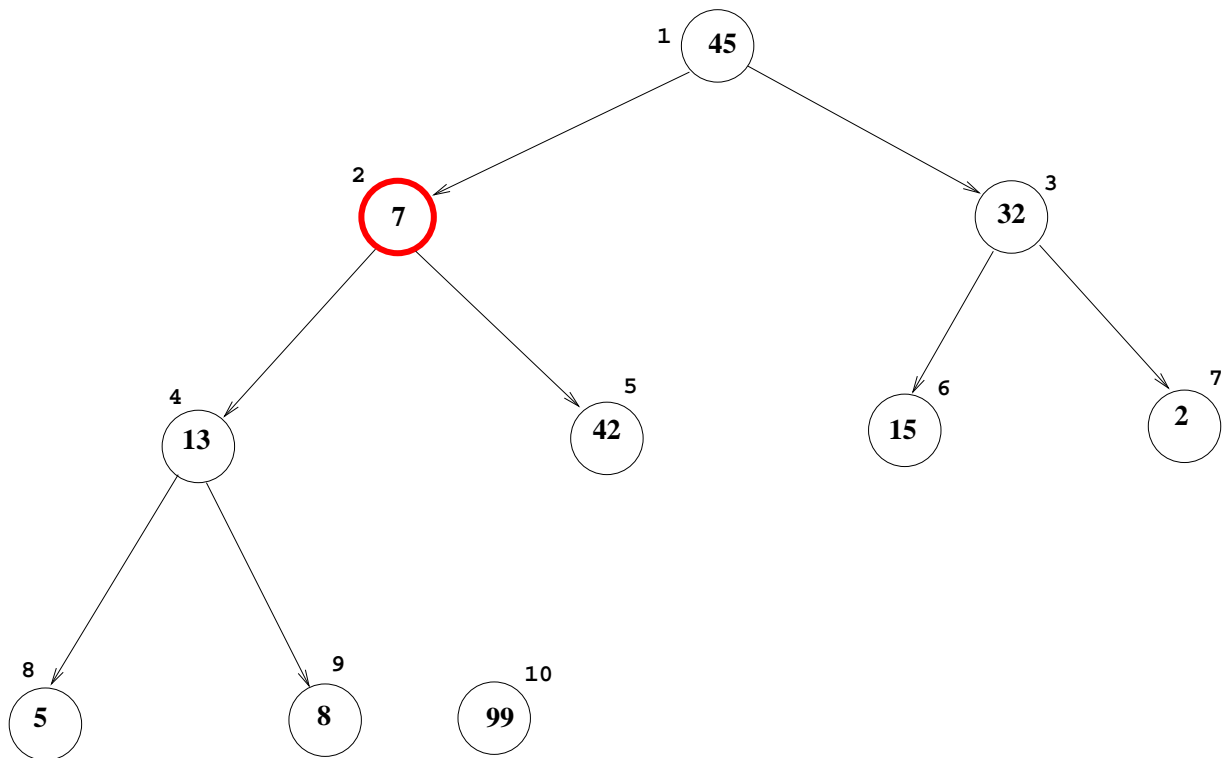
## Heapsort

- Swap the root and the node in the last entry.
- Remove last entry.
- Heapify the root.



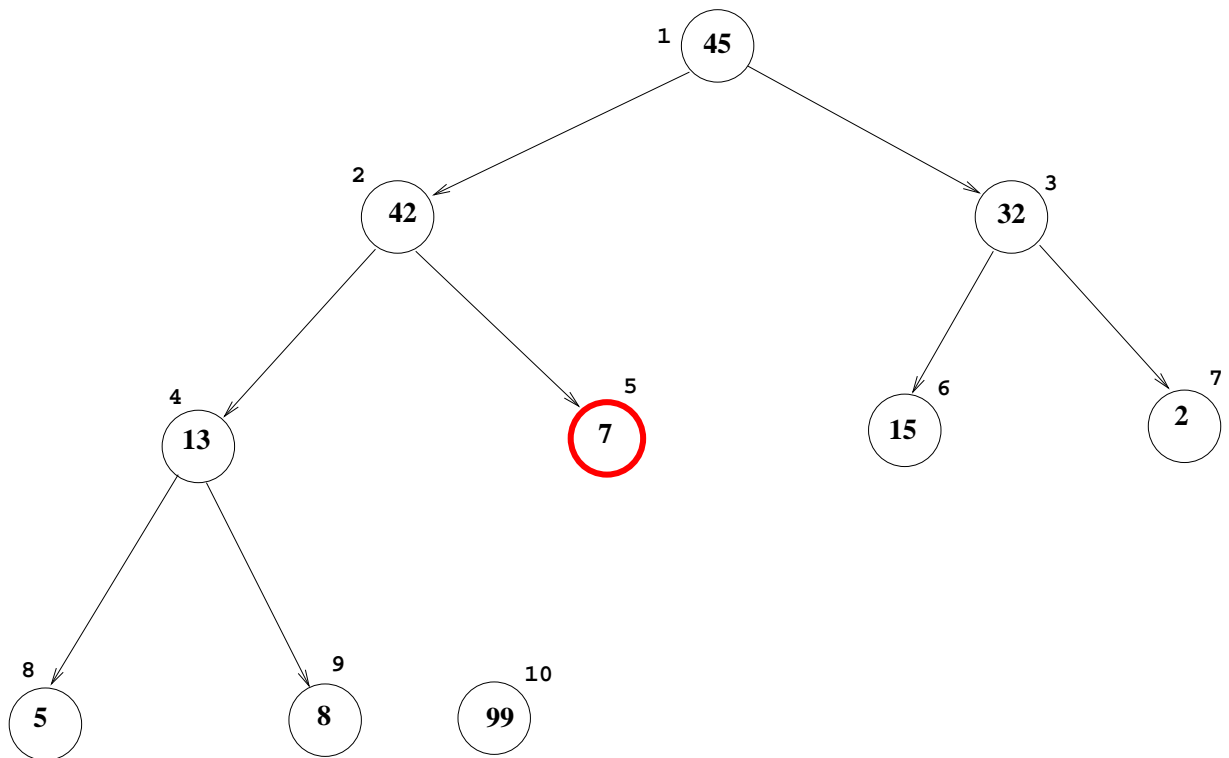
## Heapsort

- Swap the root and the node in the last entry.
- Remove last entry.
- Heapify the root.



## Heapsort

- Swap the root and the node in the last entry.
- Remove last entry.
- Heapify the root.



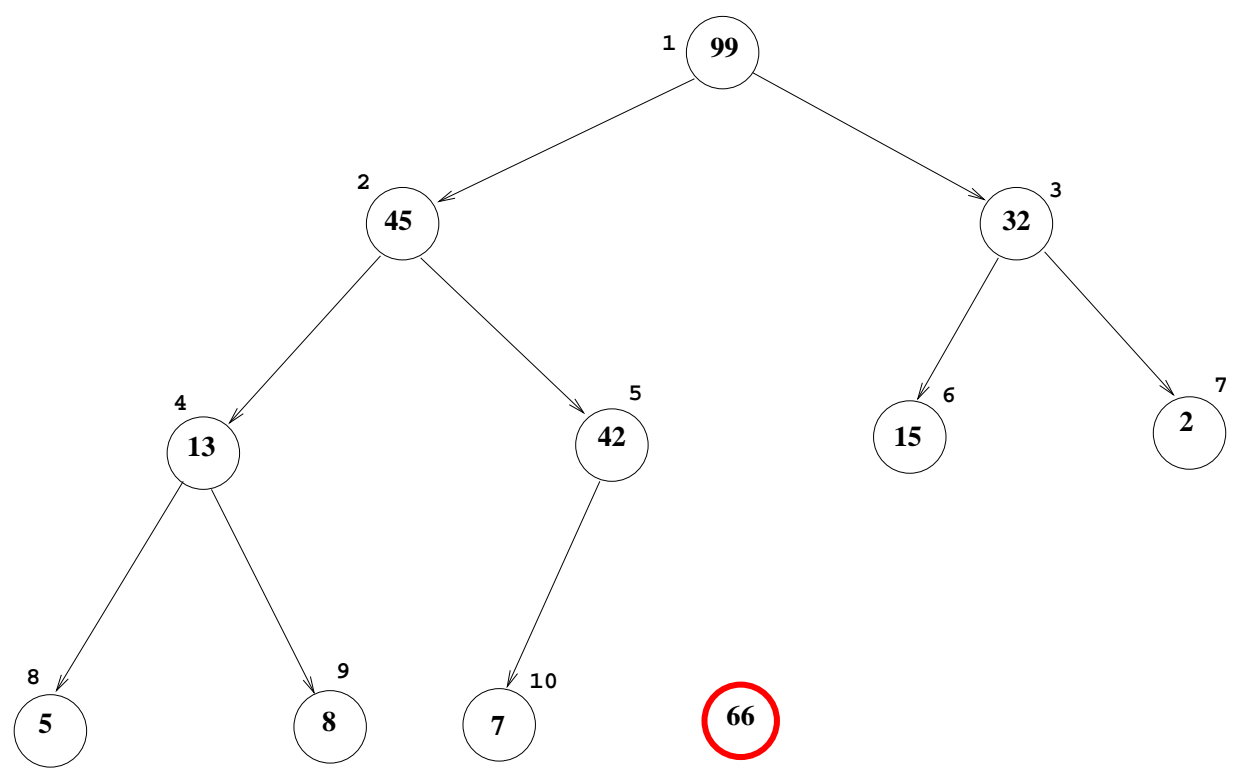
Algorithms

Sorting

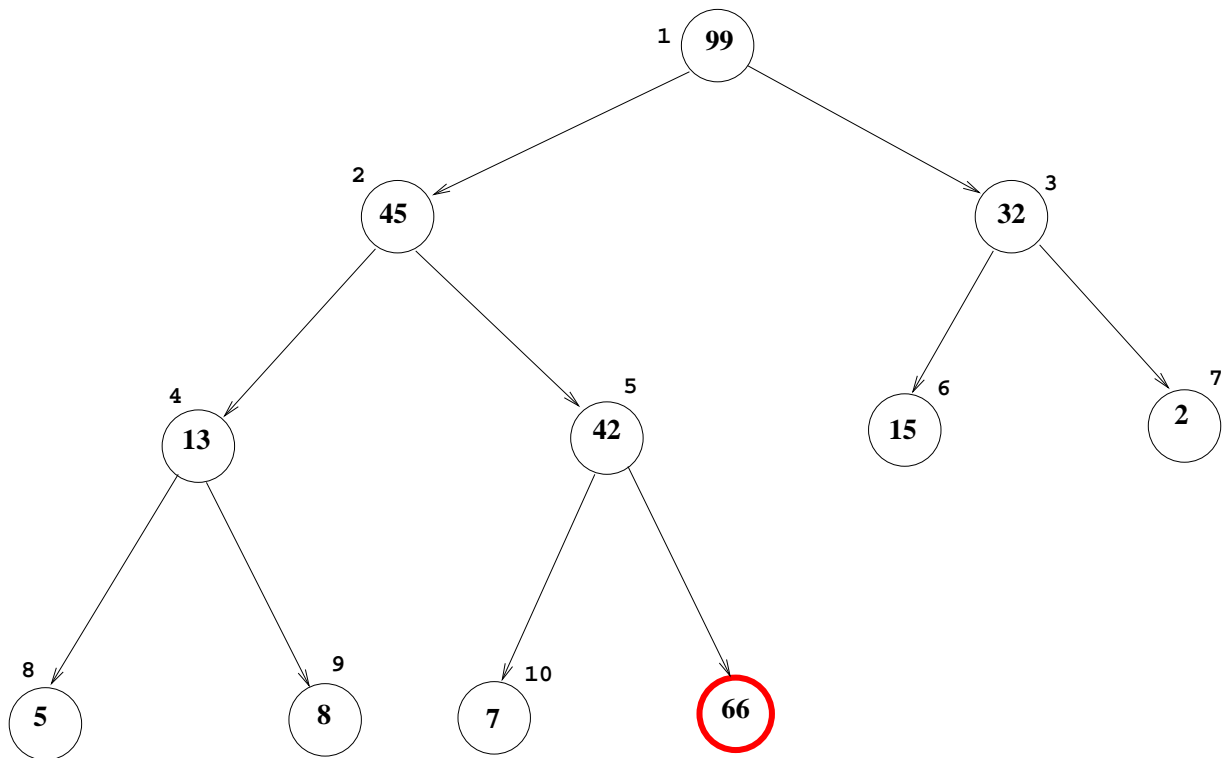
### Priority Queues

- `INSERT(S, x)`
- `MAXIMUM(S)`
- `EXTRACT-MAX(S)`

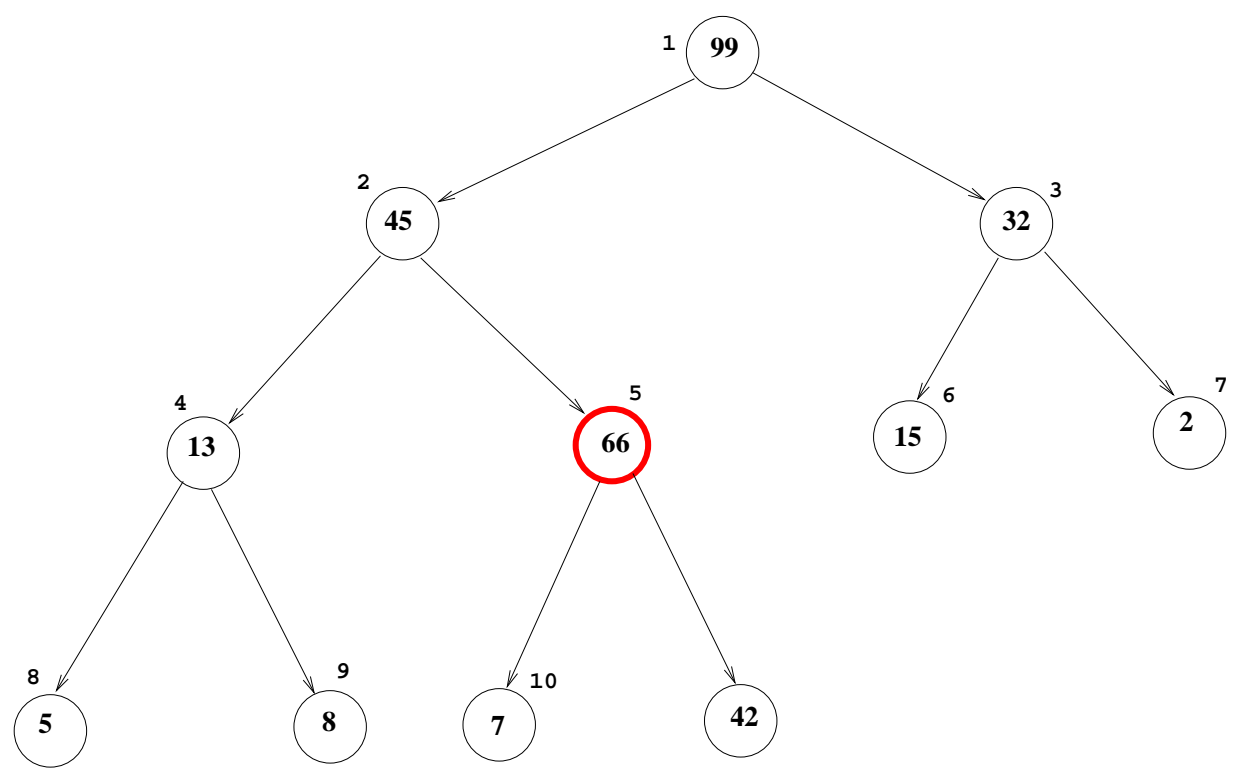
### Heap Insert



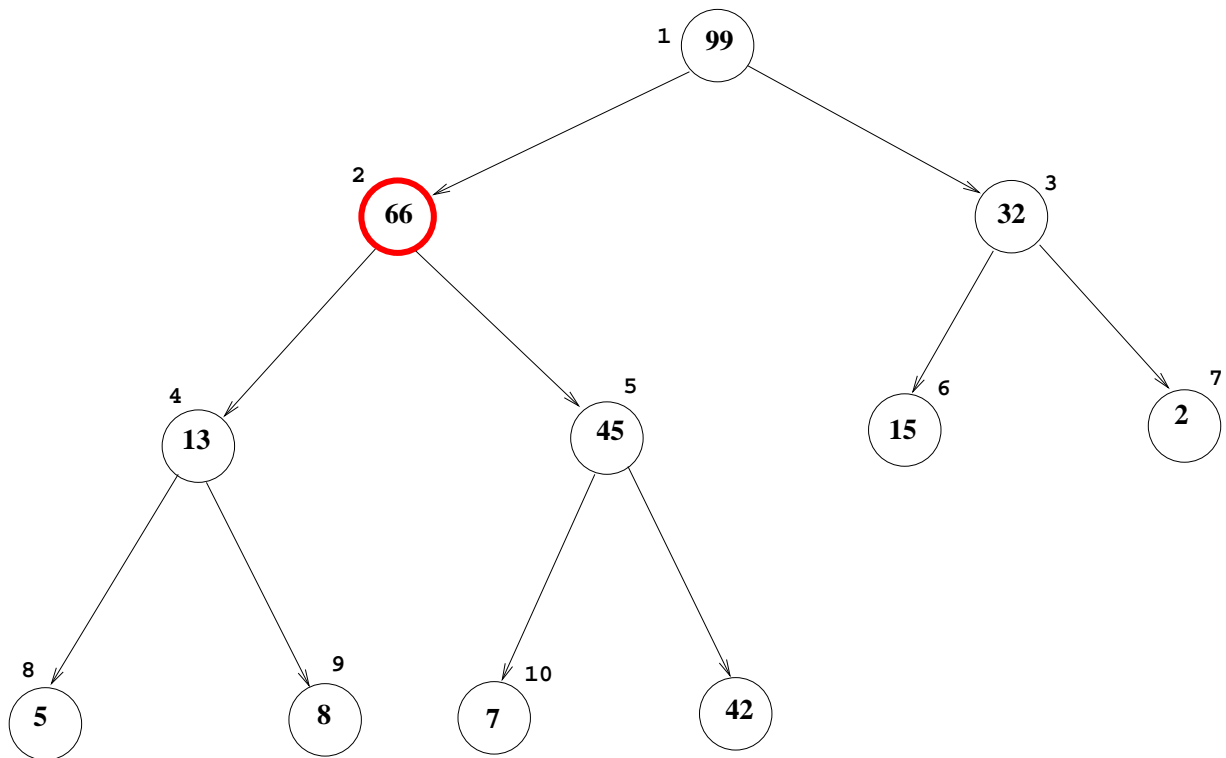
### Heap Insert



### Heap Insert



### Heap Insert



### Quicksort

- **Divide** Partition the set into two subsets such that each element in one subset is smaller than or equal to each element in the second subset.
- **Conquer** Sort each subset by using QUICKSORT recursively.
- **Combine** Do nothing!

```
QUICKSORT(A, p, r)
```

```
if p < r then  
    q = PARTITION(A, p, r);  
    QUICKSORT(A, p, q);  
    QUICKSORT(A, q+1, r);
```

Algorithms

Sorting

### Quicksort

15	45	7	13	5	42	2	99	8	32
----	----	---	----	---	----	---	----	---	----

Algorithms

Sorting

### Quicksort

32

15	45	7	13	5	42	2	99	8	
----	----	---	----	---	----	---	----	---	--

15	7	13	5	2	8	32	99	42	45
----	---	----	---	---	---	----	----	----	----

**Quicksort**

15	45	7	13	5	42	2	99	8	32
----	----	---	----	---	----	---	----	---	----

8

45

15	7	13	5	2		32	99	42	
----	---	----	---	---	--	----	----	----	--

7	5	2	8	13	15	32	42	45	99
---	---	---	---	----	----	----	----	----	----

### Quicksort

15	45	7	13	5	42	2	99	8	32
----	----	---	----	---	----	---	----	---	----

15	7	13	5	2	8	32	99	42	45
----	---	----	---	---	---	----	----	----	----

2                      15                      42                      99

7	5		8	13		32		45	
---	---	--	---	----	--	----	--	----	--

2	7	5	8	13	15	32	42	45	99
---	---	---	---	----	----	----	----	----	----

### Quicksort

15	45	7	13	5	42	2	99	8	32
----	----	---	----	---	----	---	----	---	----

15	7	13	5	2	8	32	99	42	45
----	---	----	---	---	---	----	----	----	----

7	5	2	8	13	15	32	42	45	99
---	---	---	---	----	----	----	----	----	----

5      13

2	7		8		15	32	42	45	99
---	---	--	---	--	----	----	----	----	----

2	5	7	8	13	15	32	42	45	99
---	---	---	---	----	----	----	----	----	----

### Quicksort

15	45	7	13	5	42	2	99	8	32
----	----	---	----	---	----	---	----	---	----

15	7	13	5	2	8	32	99	42	45
----	---	----	---	---	---	----	----	----	----

7	5	2	8	13	15	32	42	45	99
---	---	---	---	----	----	----	----	----	----

2	7	5	8	13	15	32	42	45	99
---	---	---	---	----	----	----	----	----	----

5

2		7	8	13	15	32	42	45	99
---	--	---	---	----	----	----	----	----	----

2	5	7	8	13	15	32	42	45	99
---	---	---	---	----	----	----	----	----	----

**Advantages of Quicksort**

- Expected running time is  $\Theta(n \log n)$
- Sorts in place.
- Quicksort: Worst-case:  $\Theta(n^2)$  (P. 149)
- Quicksort: Best-case:  $O(n \log n)$  (P. 150)
- Quicksort: Partitioning with constant proportionality:  $\Theta(n \log n)$

**Randomized Quicksort**

- Randomly permute the elements, or
- Select the partitioning number in a randomly chosen entry.

```
RANDOMIZED-PARTITION(A, p, r)
```

```
i=RANDOM(p, r);  
swap(A, r, i);  
return PARTITION(A, p, r);
```

```
RANDOMIZED-QUICKSORT(A, p, r)
```

```
if p < r then  
    q=RANDOMIZED-PARTITION(A, p, r);  
    RANDOMIZED-QUICKSORT(A, p, q);  
    RANDOMIZED-QUICKSORT(A, q+1, r);
```

**Randomized Quicksort - Analysis, Worst-Case**

- $T(n) = \max_{0 \leq q \leq n-1} \{T(q) + T(n - q - 1)\} + \Theta(n)$
- Substitution method: Guess:

$$T(n) \leq cn^2$$

$$\begin{aligned} T(n) &\leq \max_{0 \leq q \leq n-1} \{cq^2 + c(n - q)^2\} + \Theta(n) \\ &= c \max_{0 \leq q \leq n-1} \{q^2 + (n - q - 1)^2\} + \Theta(n) \\ &= cn^2 - 2c(n - 1) + \Theta(n) \\ &\leq cn^2 \end{aligned}$$

- since  $c$  can be chosen so that  $2c(n - 1)$  dominates  $\Theta(n)$ .

**Randomized Quicksort - Analysis, Average-Case**

- It is shown in the book (P. 156-158) that  $T(n) = O(n \log n)$ , i.e., average running time of QUICKSORT is  $O(n \log n)$

### Counting Sort

- Count the number of elements smaller than each given element.
- Elements to be sorted are integers between 1 and  $k$ .
- Requires  $O(k)$  working space.
- $O(k + n)$  time.
- Stable.

Algorithms

Sorting

### Radix Sort

- Sort digit for digit?
- Most significant digit first?
- Least significant digit first?

**Radix Sort**

- Sort digit for digit?
- Most significant digit first?
- Least significant digit first?

	*	*	*
289	142	142	142
488	289	234	234
142	234	289	289
441	321	321	321
234	488	432	432
478	441	441	441
445	478	445	445
321	445	478	475
475	475	475	478
432	432	488	488

**Radix Sort**

- Sort digit for digit?
- Most significant digit first?
- Least significant digit first?

	*	*	*
289	142	142	142
488	289	234	234
142	234	289	289
441	321	321	321
234	488	432	432
478	441	441	441
445	478	445	445
321	445	478	475
475	475	475	478
432	432	488	488

	*	*	*
289	441	321	142
488	321	432	234
142	142	234	289
441	432	441	321
234	234	142	432
478	445	445	441
445	475	475	445
321	488	478	475
475	488	289	478
432	289	488	488

### Bucket Sort

- Divide the interval into  $n$  equal sized subintervals and distribute the elements into these subintervals. Use insertion sort on each of  $n$  buckets.

### Summary

- Sorting by comparison requires at least  $\Omega(n \log n)$ .
- There are sorting algorithms by comparison that are optimal.
- Heapsort is optimal and sorts in-place.
- Quicksort is  $\Theta(n^2)$  but  $O(n \log n)$  on average. Small constants, faster even for quite large problem instances.
- Additional assumptions about input or stronger computational models make it possible to sort in  $\Theta(n)$  time.