Greedy Algorithms

Lecture 4

Timothy Kim
GWU CSCI 6212

Quiz

- Let QuickSortSlow be a quick sort algorithm that uses a custom ChosePivot subroutine. The custom ChosePivot subroutine picks the best possible pivot but at $O(n^2)$ time. What would be the average runtime of QuickSortSlow?

- Master Method

$$T(n) \leq a \cdot T(\frac{n}{b}) + O(n^d)$$

$$\begin{aligned} T(n) &= O(n^d \log n) && \text{if } a = b^d \\ T(n) &= O(n^d) && \text{if } a < b^d \\ T(n) &= O(n^{\log_b a}) && \text{if } a > b^d \end{aligned}$$

Sorting Lower Bound

- Theorem: Every comparison-based sorting algorithm has the worst-case running time of $\Omega(n \log n)$
- Comparison Sort: Sorting algorithm that only reads the list elements through a comparison operation that determines which of two elements should occur first in the final sorted list
- Non-comparison Sort: Bucket sort, Radix sort, and more.

Proof

- Fix a comparison-based sorting algorithm and an array of length n
- Given the input array, there are n! possible input
- Suppose that the algorithm makes $\leq$ k comparisons to correctly sort any one of the n! inputs.
- Across all n! possible inputs algorithm exhibits $\leq 2^k$ distinct possible execution.

Proof continued

- By the pigeon hole principle
  - If $2^k < n!$, there must be two distinct input that share the same execution of the algorithm
  - Therefore $2^k \geq n! \geq (n/2)^{n/2}$
  - $k \geq (n/2) \log_2(n/2) = \Omega(n \log n)$

Greedy Algorithm

- Definition: iterative process where decision is made at the moment without access to additional information about the future nor the past.
- Fractional Knapsack
- Scheduling Algorithm
- Minimum Spanning Tree (Shortest Path)
- Maybe more

Typical Structure

```
Greedy( S ):
  P = []                       // processed
  while (solutionFound(P)):    // usually is S empty
    i = selection(S)           // usually best choice
    P.add(i)
    S.remove(i)
```

D&C vs Greedy

	Divide and Conquer	Greedy
Design	Hard (finding sub-problem)	Easy
Analysis	Hard (recursion)	Easy
Correctness	Easy to establish (induction)	Hard

Proofs of Correctness

- By Induction - "greedy stays ahead"
- Exchange argument
    - By Contradiction
    - By Construction
- And more

Fractional Knapsack

- Input

$$\text{Items} = \{(w_1, v_1), (w_2, v_2), ..., (w_n, v_n)\} | w_i > 0, v_i > 0$$

$$W = \text{weight limit}$$

- Output

$$\{x_1, x_2, ..., x_n\} | 0 \leq x_i \leq 1$$

- such that we maximize $\sum_{i=1}^{n} x_i v_i$ while keeping $\sum_{i=1}^{n} x_i w_i \leq W$

Design

- Start with largest value
  - Sort by value
  - Gather until full
- Start with smallest weight
  - Sort by weight
  - Gather until full

Gather by value

```
GreedyByValue(items, W):
  X = []
  sort items by value
  for item in items:
    space = W - weight(X)
    if (space > item.weight):
      X += 1
    else:
      X += (space / item.weight)
  return X
```

Is it really optimal?

```
w = {1,3}
v = {1,3}
W = 2
X1 = GreedyByValue((w, v), W)
X2 = GreedyByWeight((w, v), W)
```

```
w = {1,3}
v = {3,1}
W = 2
X1 = GreedyByValue((w, v), W)
X2 = GreedyByWeight((w, v), W)
```

Objective Function

- Ranking function that rewards low weight and high value?

Better method

- Use ratio
  - Score $= \dfrac{v_i}{w_i}$ (higher the better)

Proof

- Let X be our greedy solution by ratio with the numbering based on score
- Let Y be the optimal solution better than X
- Let j be the point at which X and Y must differ
- Then $X_j = 1$ and $Y_j < 1$
- By swapping X_j and Y_j, we create Y' more optimal than Y
  - *(Further justification needed)*
- Contradiction, thus there can't be any solution more optimal than X

Scheduling Problem

- Given a shared resource, there are many jobs that require it. Find the optimal sequence of jobs.

- Input

$$l_j = \text{length of job } j$$
$$w_j = \text{weight or priority of job } j$$

- Output
  - Order the jobs to $\text{minimize } C = \sum_{j=1}^{n} w_j c_j$
  - where

$$c_j = \text{completion time of job } j$$

Example

- 3 jobs with length $l_1 = 1$, $l_2 = 2$ and $l_3 = 3$
- With the weight of 1, 2, and 3
- Job order = [1, 2, 3]
- then $c_1 = 1$, $c_2 = 3$, $c_3 = 6$
- Total cost = ??

Objective Function

- What should we reward?
    - weight?
    - job length?

Scheduling Algorithm

```
Schedule(L, W):
 // Sort the job weight/length
 // as the objective function
 // Homework
```

Proof of Correctness

- **Claim**: Greedy algorithm with w_j/l_j as the ordering produces the optimal job schedule.
- Proof by contradiction
  - Assume distinct score value for now.
  - Relabel the jobs in the order of our Greedy Algorithm's schedule:

$$\frac{w_1}{l_1} > \frac{w_2}{l_2} > \dots > \frac{w_n}{l_n}$$

1 2 ... n

Proof of correctness

- Let X be the schedule that our Greedy algorithm outputs
- Let Y be the optimal schedule that's not X
- Then in Y, there must be a job schedule i, j where $i > j$.
- We swap the position of job i and j to produce schedule Y'

Proof of correctness

- By swapping,
  - c_i went up by value of l_j , therefore C went up by $w_i l_j$
  - c_j went down by value of l_i therefore C went down by $w_j l_i$
- Note
  - $$i > j \rightarrow \frac{w_i}{l_i} < \frac{w_j}{l_j}$$

    $$w_i l_j < w_j l_i$$
  - This implies that Y' is better than Y, which is contradiction

Proof of correctness

- For the case where scores are NOT distinct
  - Proof will be the same until we compare Y and Y'
  - When transforming Y into Y' you get $w_i l_j \leq w_j l_i$
  - Which implies Y' is as good as Y or better.
  - Note Y' is similar to X, so we can keep swapping to transform Y into X and it'll continue to be as good or better.
  - Therefore X must be optimal.

Heap

- **max-heap**: A complete binary tree, where all nodes has key value that are greater than or equal to each of its children.
- **min-heap**: A complete binary tree, where all nodes has key value that are less than or equal to each of its children.

Heap Operations

- **Insert**: add a new object to a heap - O(log n)
- **Extract Min**: remove a node in with a minimum key value - O(log n)
- **Heapify**: n batched inserts - O(n)
- **Delete**: remove a node - O(log n)

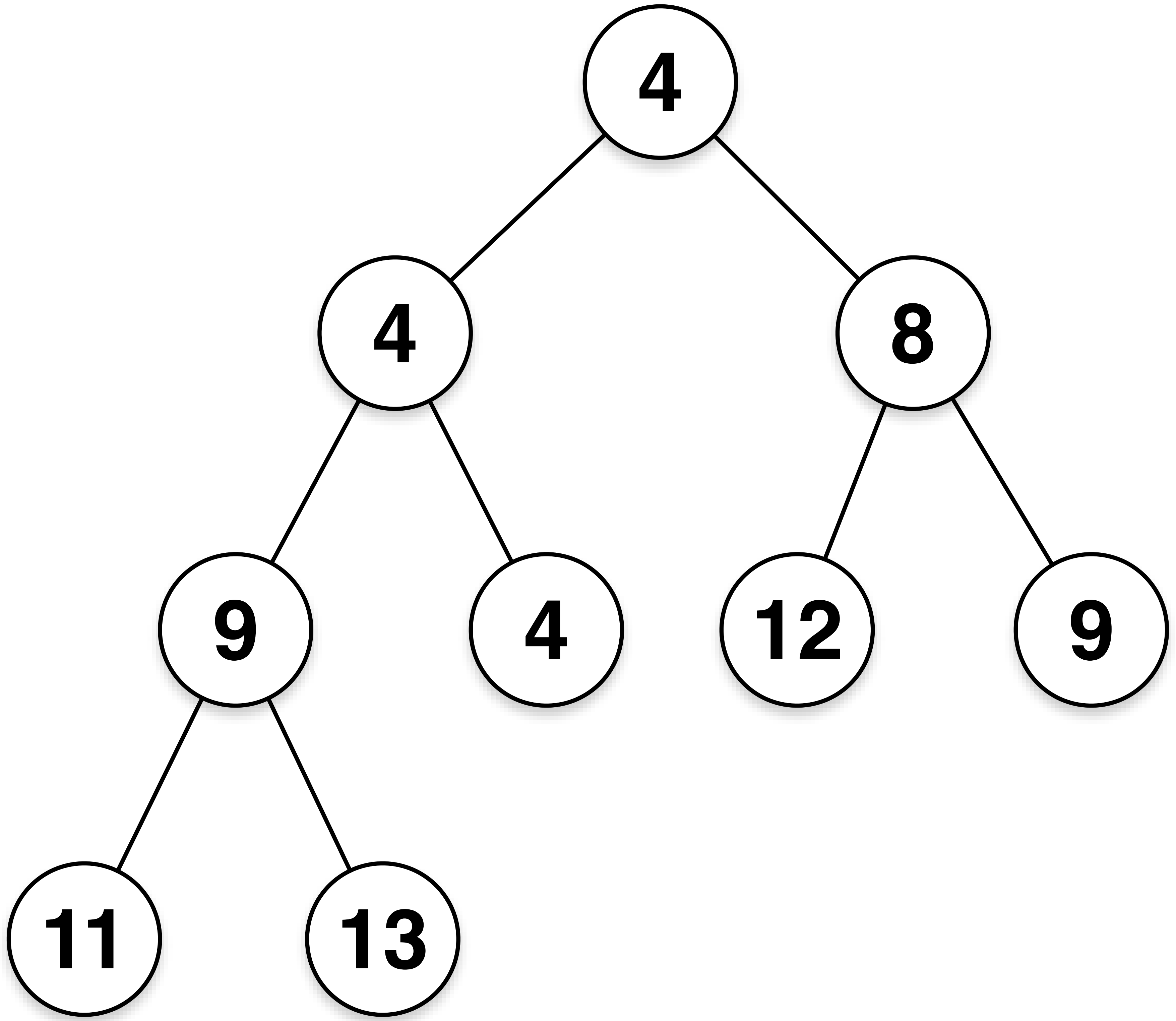
Application

- If your application requires minimum calculation repeatedly
    - **Selection Sort**: find min O(n^2)
    - **Heap Sort**: same as selection sort but O(n log n)
        - Heapify input array - O(n)
        - Extract-min n times - O(n log n)
- Event manager - "priority queue"
- Median maintenance - extract median

Implementation

- Usually trees are implemented using pointers and references. However because Heaps are complete binary tree, we can use an array.
- root node: 1^{st} element
- $\text{child}(i) = (2i, 2i + 1)$
- $\text{parent}(i) = \begin{cases} i/2 & \text{if } n \text{ is even} \\ \lfloor i/2 \rfloor & \text{if } n \text{ is odd} \end{cases}$

Implementation



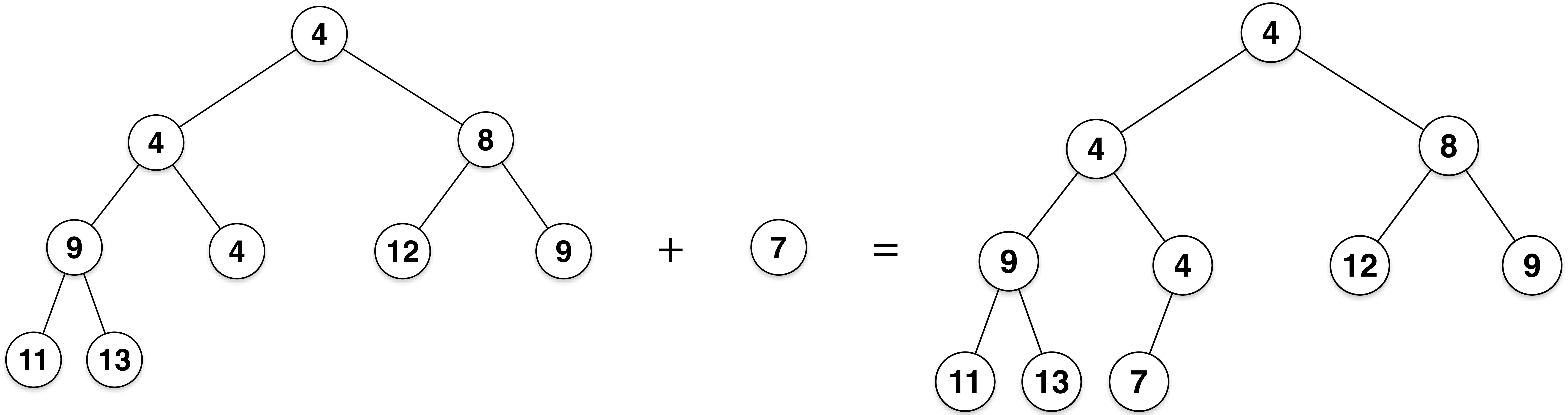
1	2	3	4	5	6	7	8	9
4	4	8	9	4	12	9	11	13

Insert

```
Insert(k):
  Add k to the next place (append)
  if (parent(k).key < k.key):
    done
  else:
    heapify-up (bubble up)
```

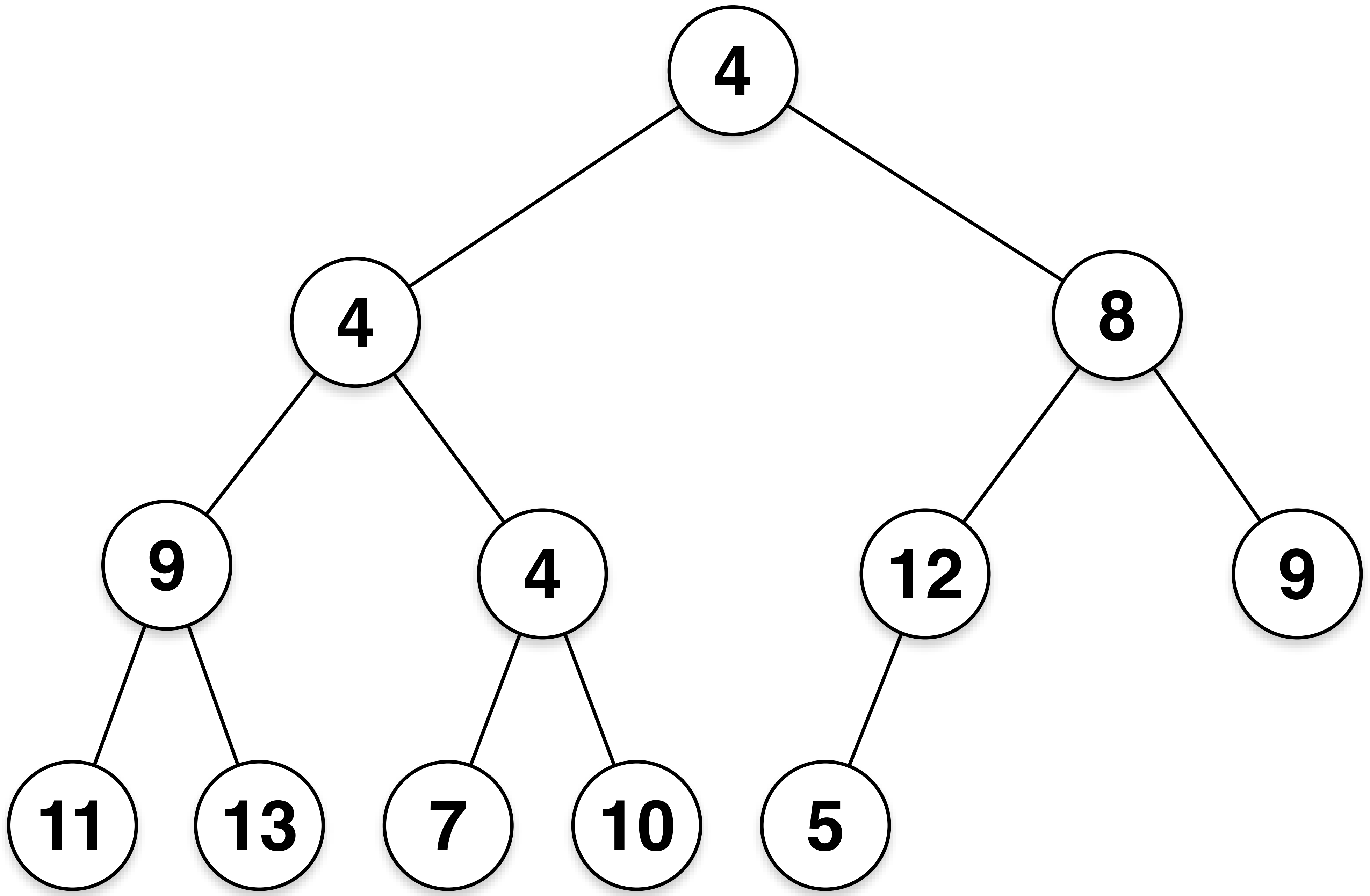
- **Invariant**: all nodes has key value that are less than or equal to each of its children

Example

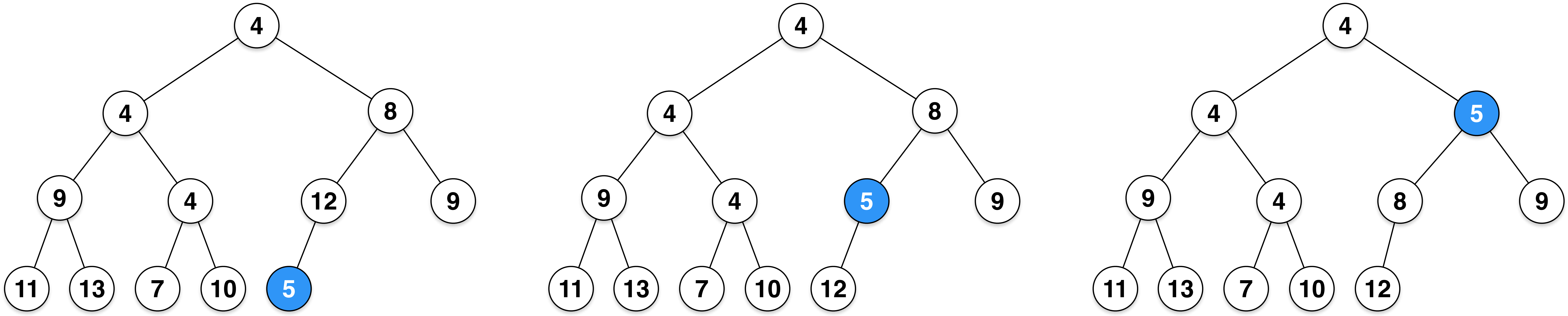


Heapify Up

```
HeapifyUp(A, k):
  while(parent(k).key > k.key):
    swap(parent(k), k)
```



Heapify Up



Insert Running Time

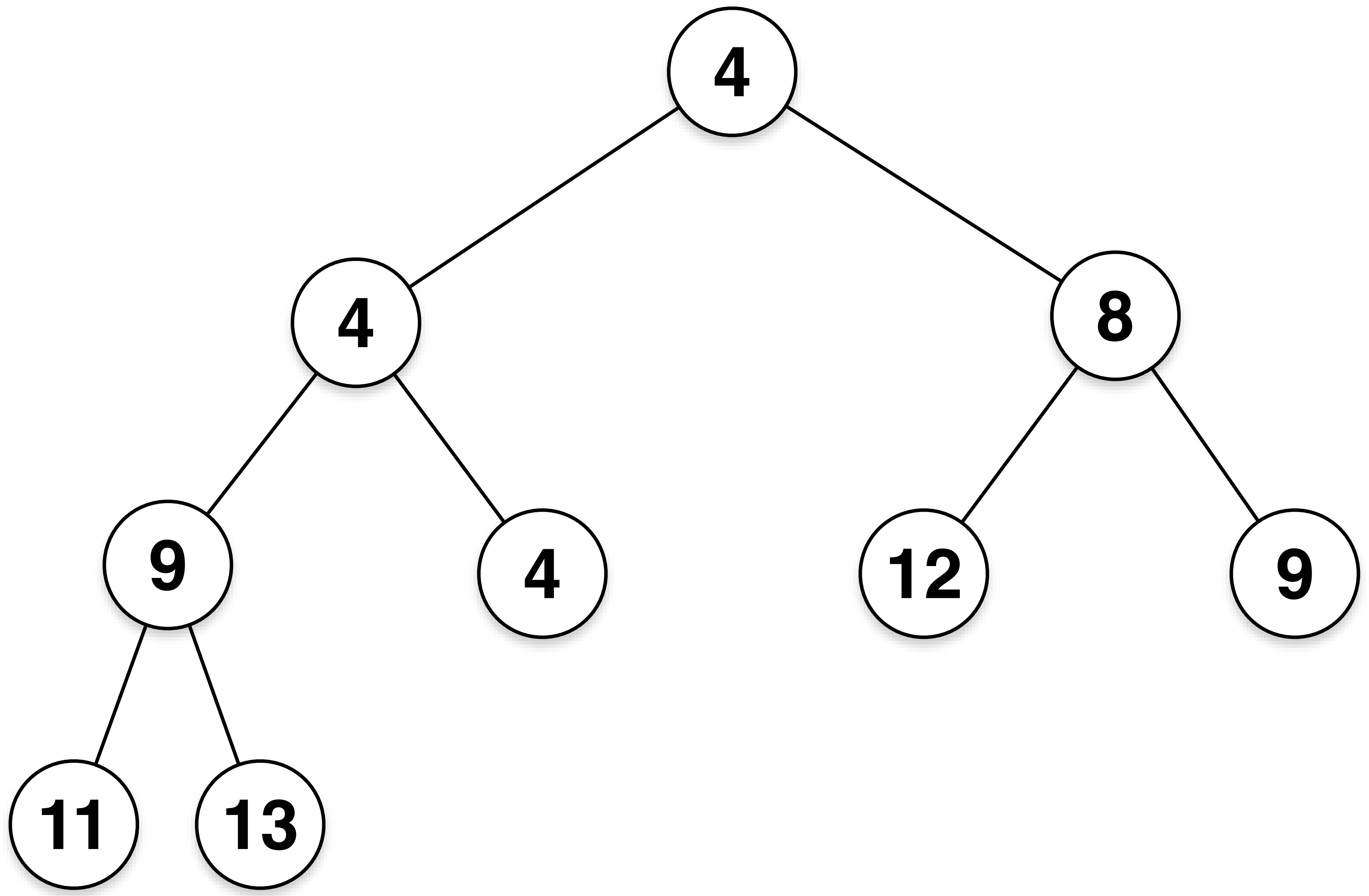
Extract Min

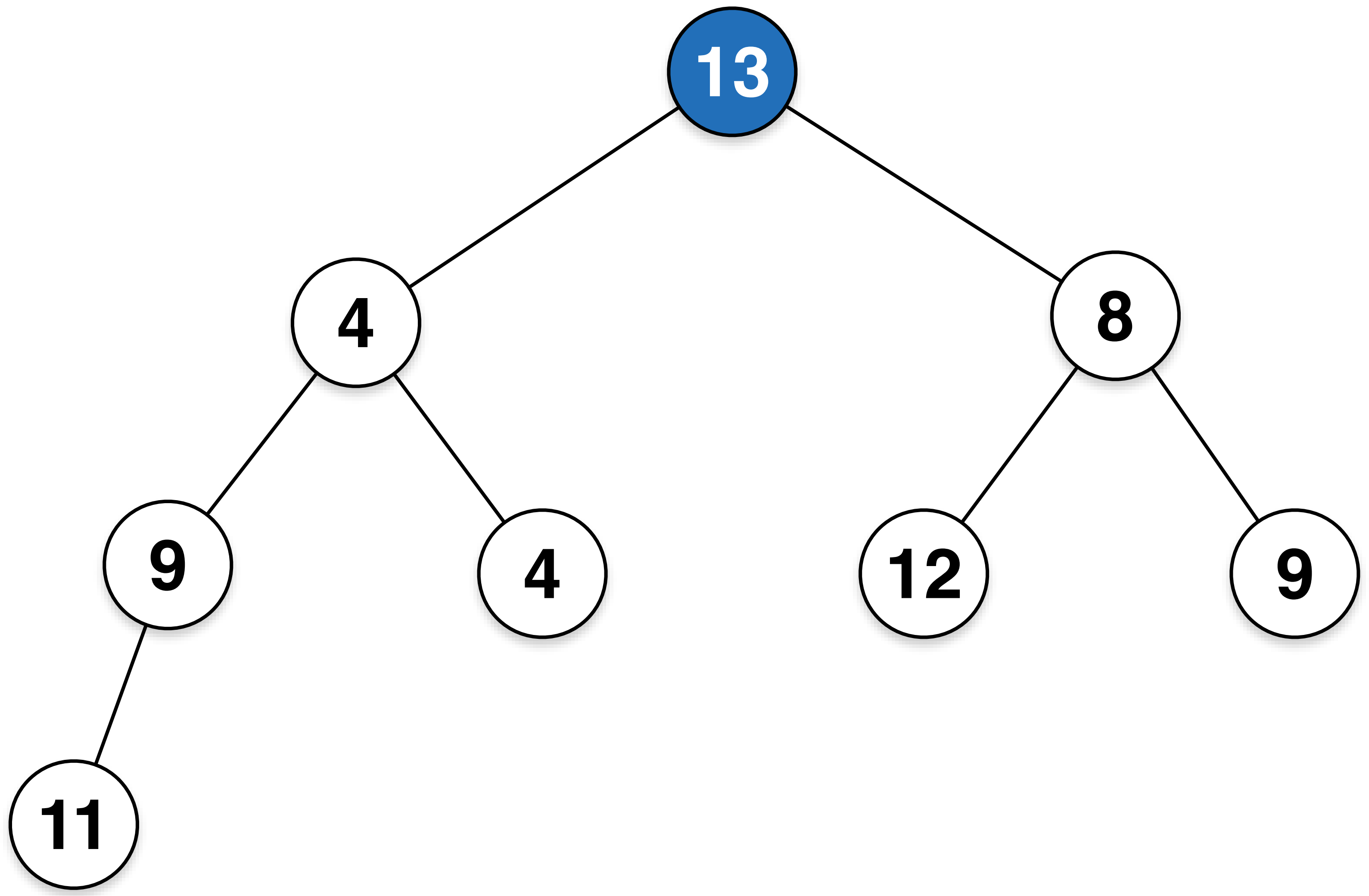
```
ExtractMin(A):
  swap root with the last element
  remove previous root (the min)
  MinHeapify(A, 1)
  return previous root (the min)
```

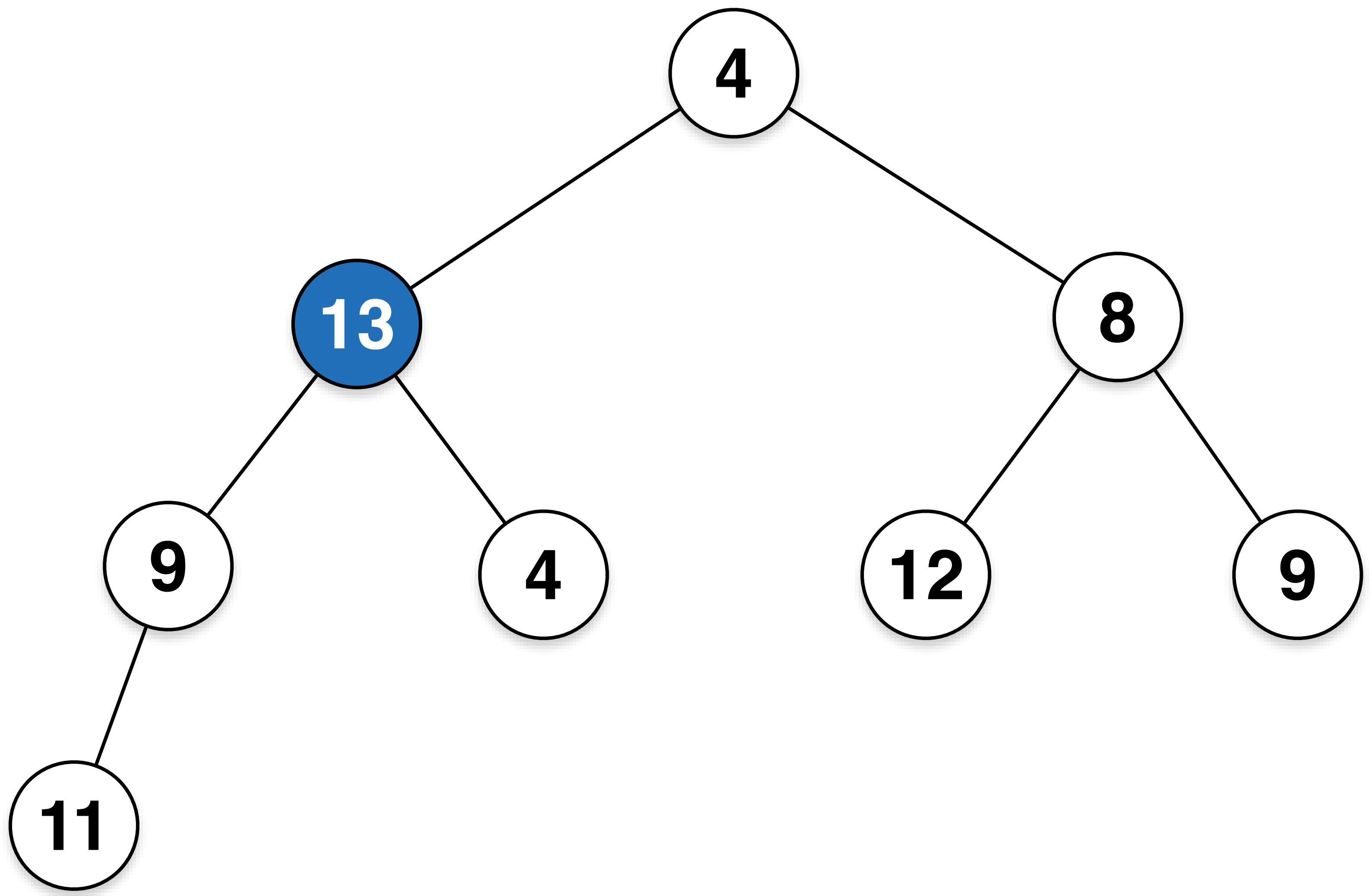
MinHeapify

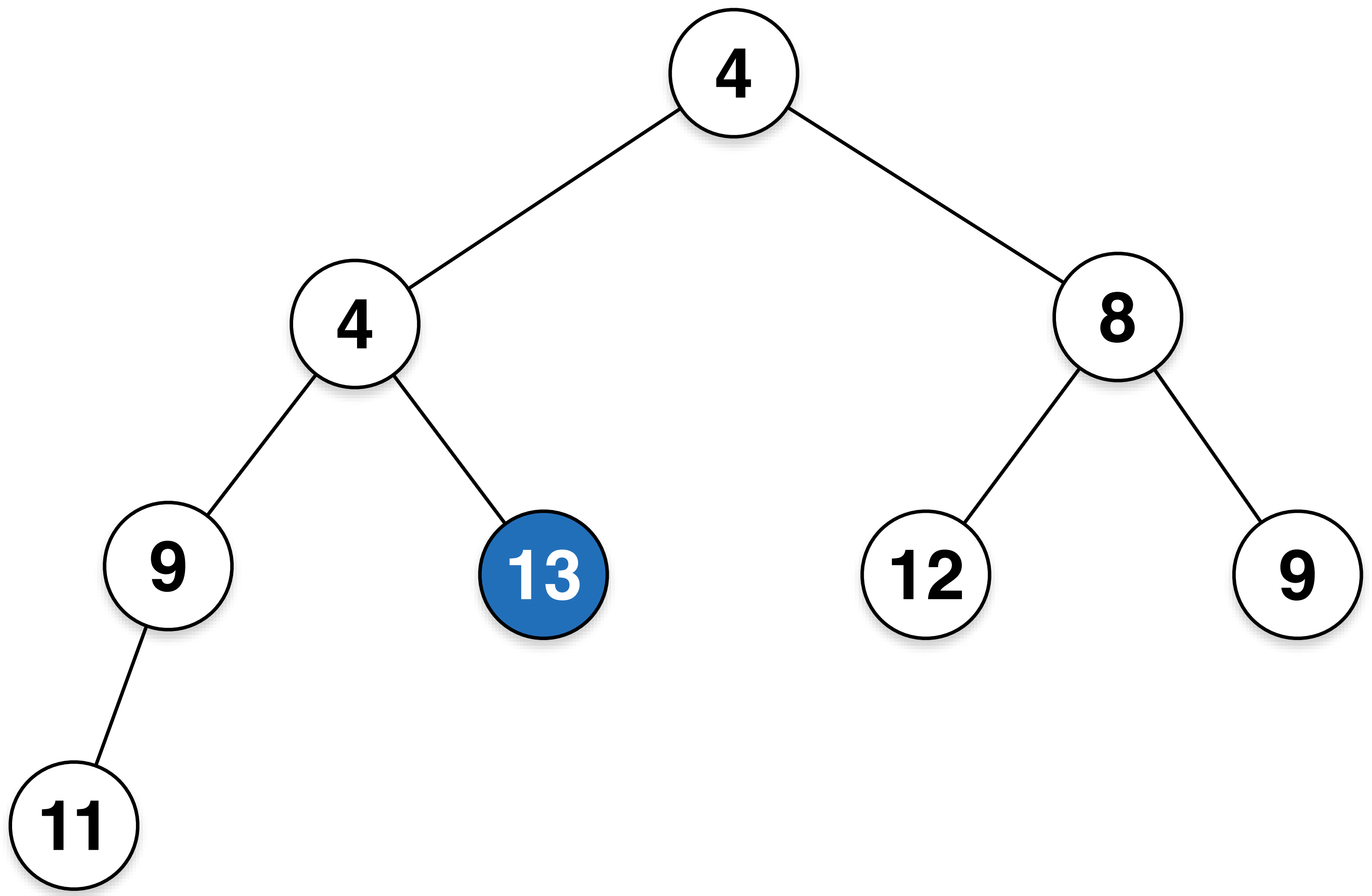
```
MinHeapify(A, i):
  left = 2i
  right = 2i + 1
  min = i
  if (A[min] < A[left] && A[min] < A[right])
    return
  if (A[left] < A[right]):
    min = left
  else:
    min = right
  swap(A[i], A[min])
  MinHeapify(A, min)
```

MinHeapify









Heapify

```
Heapify(A):
  Assume A is already a heap
  start = floor(n/2) // first root with children
  for i = start to 1:
    MinHeapify(A, i)
```

Heapify Running Time

- $O(\log\, n)$??
- If all the subtree at height h has been heapified, then heapifying the sub tree at h+1 level will only require bubbling down the root nodes.
    - $O(h)$ operations (swap) per node
    - Height is measured from bottom up starting at 0
- Notice how most of the heapifying happens at the bottom.

Heapify Running Time

- $O(h)$ operations (swap) per node
- $\text{NodeCount}(h) = \lceil 2^{(\log_2 n - h) - 1} \rceil = \left\lceil \frac{2^{\log n}}{2^{h+1}} \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil$
- Cost of heapifying the entire tree:

$$\sum_{h=1}^{\lceil \log n \rceil} \frac{n}{2^{h+1}} O(h) = O\left(n \sum_{h=1}^{\lceil \log n \rceil} \frac{h}{2^{h+1}} \right)$$

$$\leq O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h} \right) = O(2n) = O(n)$$

- http://www.symbolab.com/solver/series-calculator/%5Csum_%7Bn%3D0%7D%5E%7B%5Cinfty%7D%20%5Cfrac%7Bn%7D%7B2%5E%7Bn%7D%7D