CS 580
ALGORITHM DESIGN AND ANALYSIS

Basics: Asymptotic Analysis
(Chapter 2 in the "Algorithm Design" book)

Vassilis Zikas

COMPUTATIONAL TRACTABILITY

- A major focus of this class is to find **efficient** algorithms
- But what does that mean?
- We will mostly focus on running time (we want algorithms that run quickly)
  - Other notions of efficiency might come up: space, number of samples, etc

WORST CASE RUNNING TIME

- We will mostly focus on <u>**worst-case**</u> running time
- Mathematically convenient
- More appealing for some applications (software for a plane!)
- Bad alternatives:
  - E.g. “Average-case analysis” is much harder mathematically and needs to assume distribution over instances
  - What’s a good distribution?

POLYNOMIAL TIME

- For many (most?) natural problems there exists a trivial algorithm: check every possible solution!
  - Typically takes time 2^n
  - Typically it's unacceptable…
- Proposed definition of efficiency: An algorithm is efficient if it achieves qualitatively better worst-case performance (at an analytical level) than brute-force search.
  - Too vague

POLYNOMIAL TIME

- The search space (typically) increases exponentially in the input size.
- A good algorithm should slow down a little bit (by a constant factor) when the input size increases a bit (by a constant factor).
- **Property:** *There exists constants* $c > 0$ *and* $d > 0$ *such that on every input size* N *the running time of the algorithm is bounded by* cN^d *steps*
- Any polynomial time bound satisfies this property
  - If the input size increases from N to $2N$ the bound increases from cN^d to $c(2N)^d = c2^dN^d = c'N^d$
  - Since d is a constant, 2^d is also a constant!
- Proposed definition of efficiency: An algorithm is efficient if it has a polynomial running time.

POLYNOMIAL TIME

- Justification: It really works in practice!
  - Although $10^{50}n^{100}$ is technically poly-time it would be useless in practice
  - But, in practice, the algorithms we do develop almost always have small constants and small exponents
  - That is, breaking through the barrier of ``brute force'' typically exposes some fundamental structure of a problem
- Exceptions:
  - There are some polytime algorithms never used in practice because they are very slow (e.g. solving a semi-definite program might fall in this category)
  - There are exponential time algorithms that are used often in practice because they are fast in real world instances (e.g. the simplex algorithm for solving LPs)

WHY IT MATTERS

Table 2.1 The running times (rounded up) of different algorithms on inputs of increasing size, for a processor performing a million high-level instructions per second. In cases where the running time exceeds 10^{25} years, we simply record the algorithm as taking a very long time.

	n	$n \log_2 n$	n^2	n^3	1.5^n	2^n	$n!$
$n = 10$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	4 sec
$n = 30$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	< 1 sec	18 min	10^{25} years
$n = 50$	< 1 sec	< 1 sec	< 1 sec	< 1 sec	11 min	36 years	very long
$n = 100$	< 1 sec	< 1 sec	< 1 sec	1 sec	12,892 years	10^{17} years	very long
$n = 1{,}000$	< 1 sec	< 1 sec	1 sec	18 min	very long	very long	very long
$n = 10{,}000$	< 1 sec	< 1 sec	2 min	12 days	very long	very long	very long
$n = 100{,}000$	< 1 sec	2 sec	3 hours	32 years	very long	very long	very long
$n = 1{,}000{,}000$	1 sec	20 sec	12 days	31,710 years	very long	very long	very long

ASYMPTOTIC ORDER OF GROWTH

- **Upper Bounds:** $T(n)$ is $O\big(f(n)\big)$, or $T(n) \in O(f(n))$, if the exists constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \leq c \cdot f(n)$
  - People sometimes write $T(n) = O(f(n))$ but that's slight abuse of notation
- Example: $g(n) = 32n^2 + 17n + 30$
  - $g(n)$ is $O(n^2), O(n^3), O(2^n), \ldots$
  - $c = 100, n_0 = 1$: $g(n) \leq 100 \cdot n^2$ for all $n \geq 1$
  - $g(n)$ is not $O(n), O(n\ logn)$

ASYMPTOTIC ORDER OF GROWTH

- **Lower Bounds:** $T(n)$ is $\Omega\big(f(n)\big)$, or $T(n) \in \Omega(f(n))$, if the exists constants $c > 0$ and $n_0 \geq 0$ such that for all $n \geq n_0$ we have $T(n) \geq c \cdot f(n)$.
- **Tight Bounds:** $T(n)$ is $\Theta\big(f(n)\big)$, or $T(n) \in \Theta(f(n))$, if it is both $O(f(n))$ and $\Omega(f(n))$.
- Example: $g(n) = 32n^2 + 17n + 32$
  - $g(n) \in \Omega(n^2)$
  - $c = 1, n_0 = 1$: $g(n) \geq n^2$ for all $n \geq 1$
  - Since $g(n) \in O(n^2)$ as well we have that $g(n) \in \Theta(n^2)$

NOTATION

- Be careful!
  - $f(n) = n^3, g(n) = n^2$
  - $f(n) = O(n^3), g(n) = O(n^3)$
  - But, $f(n) \neq g(n)$
  - Writing " $=$ " can be confusing
- Weird statements: "Any comparison-based sorting algorithm takes $O(nlogn)$ steps"
  - Statement doesn't make sense…
  - For lower bounds we use Ω

PROPERTIES

- Transitivity
  - If $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$
  - *Proof*:
  - Let c and n_0 be constants such that $f(n) \leq cg(n)$ for all $n \geq n_0$. Also, let c' and $n_0{}'$ be constants such that $g(n) \leq c'h(n)$ for all $n \geq n_0{}'$.
  - $f(n) \leq cg(n) \leq c \cdot c' \cdot h(n)$ for all $n \geq \max\{n_0, n_0{}'\}$.

PROPERTIES

- Transitivity
  - If $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$
  - If $f \in \Omega(g)$ and $g \in \Omega(h)$ then $f \in \Omega(h)$

- Additivity
  - If $f \in O(g)$ and $g \in O(h)$ then $f + g \in O(h)$
  - If $f \in \Omega(g)$ and $g \in \Omega(h)$ then $f + g \in \Omega(h)$
  - If $f \in \Theta(h)$ and $g \in \Theta(h)$ then $f + g \in \Theta(h)$

ASYMPTOTIC BOUNDS FOR COMMON FUNCTIONS

- Polynomials
  - $a_0 + a_1 n + \cdots + a_d n^d$ is $\Theta(n^d)$ if $a_d > 0$.
- Logarithms
  - $\log_a n$ is $\Theta(\log_b n)$ for any constants $a, b > 1$
    - That is, the base of the logarithm doesn't really matter
  - For every $x > 0$ and every $b > 1$, $\log_b n \in O(n^x)$
    - That is, every logarithm is faster than every polynomial
- Exponentials
  - For every $r > 1$ and every $d > 0$, n^d is $O(r^n)$
    - That is, every exponential is slower than every polynomial

SURVEY OF COMMON RUNNING TIMES

- Styles of analysis recur frequently and lead to similar bounds
  - That ‘s why we see $O(n), O(nlogn)$ and $O(n^2)$ all the time

LINEAR TIME

- Running time is proportional to the size of the input
- Example 1: Compute the maximum of n positive numbers

$$temp = 0$$

$$for\ i = 1, \ldots, n:$$

$$If\ a_i > temp:$$

$$temp = a_i$$

LINEAR TIME: MERGE TWO LISTS

- Example 2: merging two sorted lists
- Let $a_1 \leq a_2 \leq \cdots \leq a_n$ and $b_1 \leq \cdots \leq b_n$ be two sorted lists on numbers
- We want $c_1 \leq \cdots \leq c_{2n}$ to be the merged output
  - E.g. 2, 3, 11 and 4, 9, 16 would give 2, 3, 4, 9, 11, 16

LINEAR TIME: MERGE TWO LISTS

Algorithm

1. $i = 1, j = 1$
2. $while\ (both\ lists\ are\ non\ empty)$:
   - $if\ a_i \leq b_j$: append a_i to output and $i += 1$
   - Else: append b_j to output and j $+= 1$
3. Append the remainder of the non-empty list to the output

Claim: The above algorithm runs in linear time

Proof: At each step of the while loop either i or j increases by 1. One of the two input lists will be empty once i or j hits n, which happens in at most $2n$ steps.

LINEAR TIME: MERGE TWO LISTS

- <u>Correct but suboptimal analysis:</u>
  - Every element is involved in at most n comparisons.
  - There are $2n$ elements
  - Running time is at most $2n^2$
- Real algorithms are not this simple
- For some advanced topics we might choose the suboptimal route in exchange for a more crisp analysis

$O(nlogn)$ TIME

- Common in divide and conquer algorithms
- Typically the running time of a solution that splits its input into two inputs, solves each piece, and then combines the solutions
- Example: The analysis we had for computing A^n in Fibonacci
- Example: Mergesort (sort n numbers)
  - Divide the input into two equal sets
  - Sort each half recursively
  - Merge sorted lists
- Very common: running time of algorithms whose most expensive step is to sort
  - E.g. given n time stamps $t_1, t_2, \dots, t_n$ (say corresponding to arrivals) find the largest interval (with no arrival)
  - Algorithm: (1) Sort the time stamps. (2) Scan the sorted list keeping track of the maximum gap between successive intervals

QUADRATIC TIME

- Common: enumerate all pairs of elements
- E.g. Given a list of n points in two dimensions $(x_1, y_1), (x_2, y_2), \ldots$ find the pair of points that is closest to each other (in, say, Euclidean distance)
  - $O(n^2)$ solution: try all pairs
  - Note: seems that $\Omega(n^2)$ is unavoidable but we can actually do better!

CUBIC TIME

- Enumerate all triplets
- E.g. Given n sets $S_1, S_2, \ldots \subseteq [n]$, is there a pair of sets that are disjoint?
  - $[n] = \{1, 2, \ldots, n\}$
  - $O(n^3)$: for each pair of subsets (n^2 of them) decide if this pair is disjoint (linear time assuming it takes constant time to check if $v \in S_i$)

POLYNOMIAL TIME $O(n^k)$

- Independent set of size k: Given a graph, is there a set of k nodes such that no two nodes are connected by an edge?
  - k here is a constant
  - $O(n^k)$ solution: enumerate all subsets of size k
  - There are $\binom{n}{k} = \frac{n(n-1)\dots(n-k+1)}{k!} \leq \frac{n^k}{k!}$ subsets
  - Checking a subset takes $O(k^2)$ steps
  - Overall $O\left(k^2 \frac{n^k}{k!}\right) = O(n^k)$

EXPONENTIAL TIME

- Independent set: Given a graph G what is the maximum size of an independent set?
- Equivalent (up-to polytime operations): Given a graph G is there an independent set of size k?
  - Here k is ***not*** a constant!
- $\mathrm{O}(\mathrm{n}^2 2^\mathrm{n})$ solution: enumerate all subsets!

SUBLINEAR TIME

- Less time than it takes to read the input!
  - How come?
  - Typically there is an assumption about the input hidden…
- E.g. Given a <u>sorted</u> list of numbers, is the number p one of them?
  - $O(\log(n))$ solution: binary search
  - Be careful though: n here is the size of the list! We are assuming a comparison takes constant time…

AN ASIDE: PRIORITY QUEUES

- Our goal: develop algorithms and algorithmic techniques
- Sometimes implementation details (e.g. good data structures) might make a big difference in terms of running time
- Today: *the priority queue*

PRIORITY QUEUES

- Maintains a set of elements S
- For each element $v \in \mathrm{S}$ there is an associated key $key(v)$ that denotes the priority of this element
  - Smaller key, higher priority
- Priority queue: support addition and deletion of elements, as well as retrieval of element with the smallest key
- Goals? How fast can we hope for things to be?
  - Note: One can use a priority queue for sorting
  - Insert each element and let $key(v) = v$
  - Then retrieve and the new list is sorted
  - So, roughly we should hope for $\log(n)$ time operations

REVIEW: HEAP DATA STRUCTURE

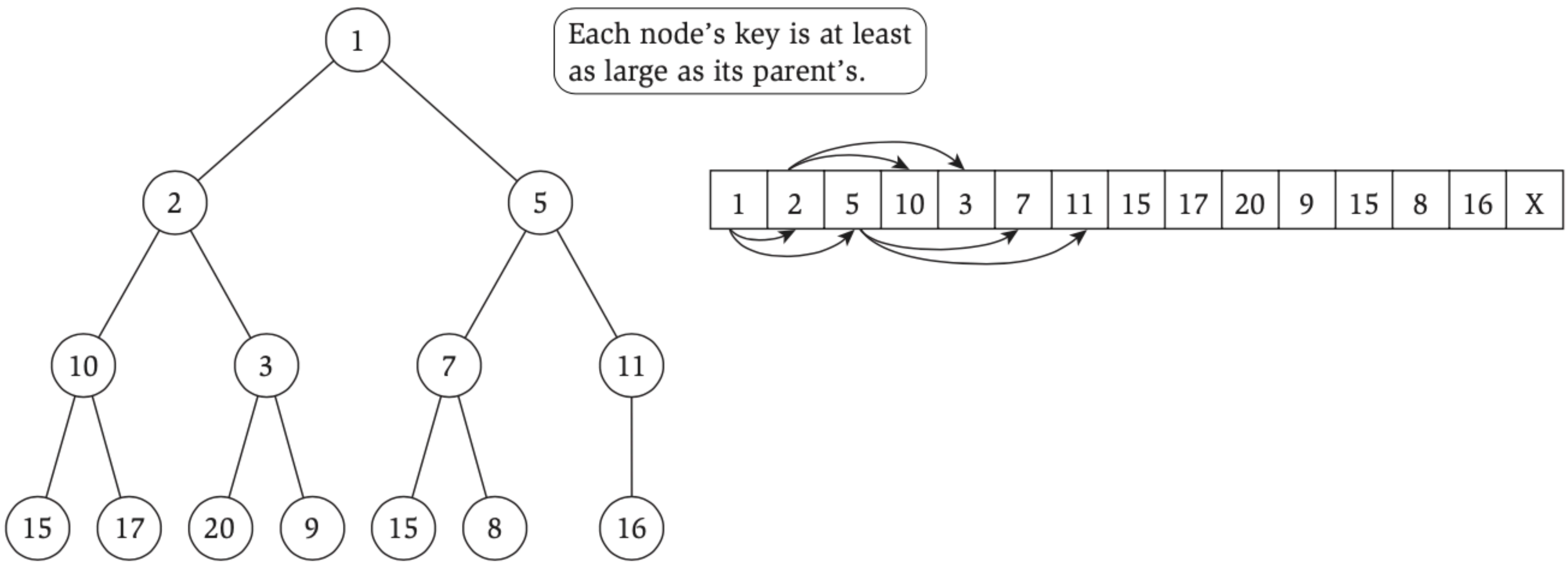
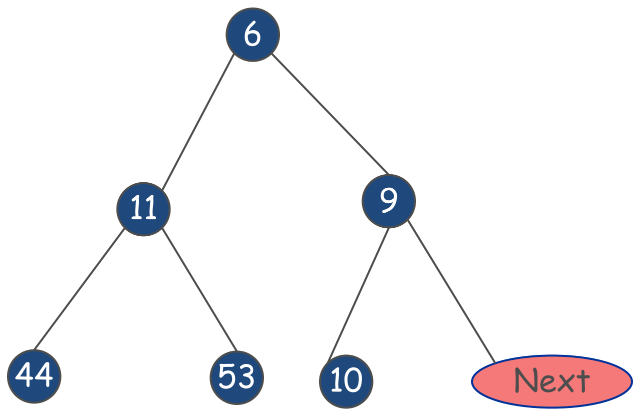


Figure 2.3 Values in a heap shown as a binary tree on the left, and represented as an array on the right. The arrows show the children for the top three nodes in the tree.

REVIEW: HEAP DATA STRUCTURE



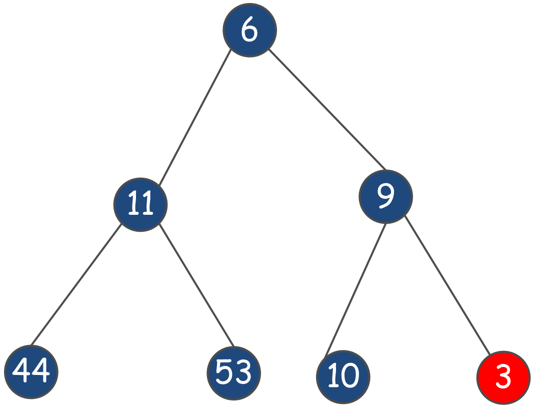
Min Heap Order: For each node v in the tree

$$\mathrm{Parent}(v).\mathrm{Value} \leq v.\mathrm{Value}$$

Max Heap Order: For each node v in the tree

$$\mathrm{Parent}(v).\mathrm{Value} \geq v.\mathrm{Value}$$

HEAP INSERTION

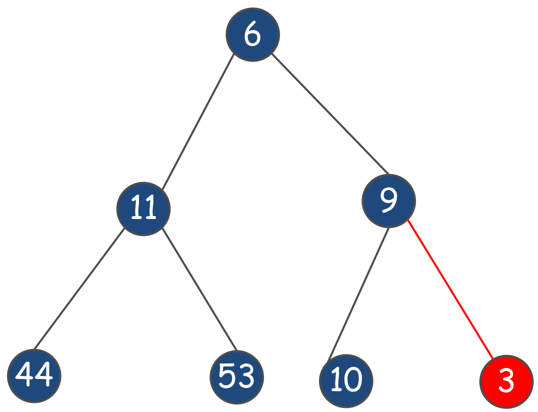


Heap.Insert(3)

Min Heap Order: For each node v in the tree

$$\mathrm{Parent(v).Value} \leq \mathrm{v.Value}$$

HEAP INSERTION



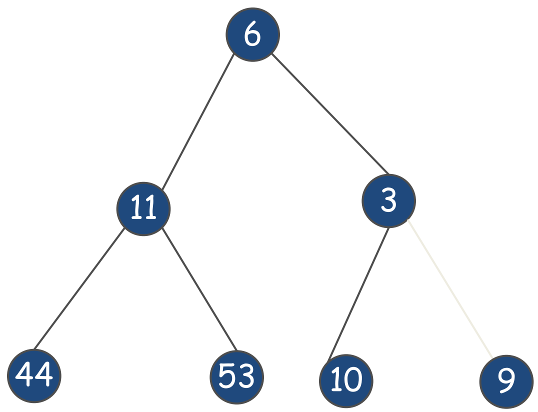
Heap.Insert(3)

Min Heap Order: For each node v in the tree

$$\text{Parent}(v).\text{Value} \leq v.\text{Value}$$

HEAP INSERTION

Heap.Insert(3)

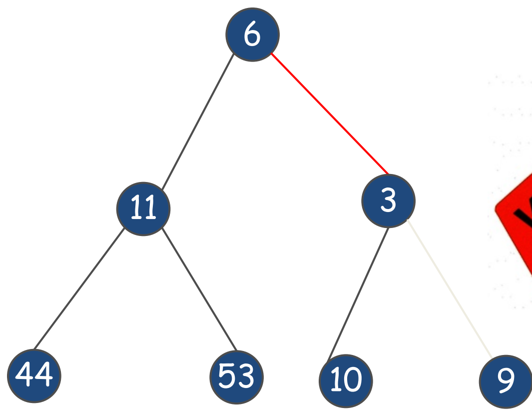


Min Heap Order: For each node v in the tree

$$\mathrm{Parent(v).Value} \leq \mathrm{v.Value}$$

HEAP INSERTION

Heap.Insert(3)

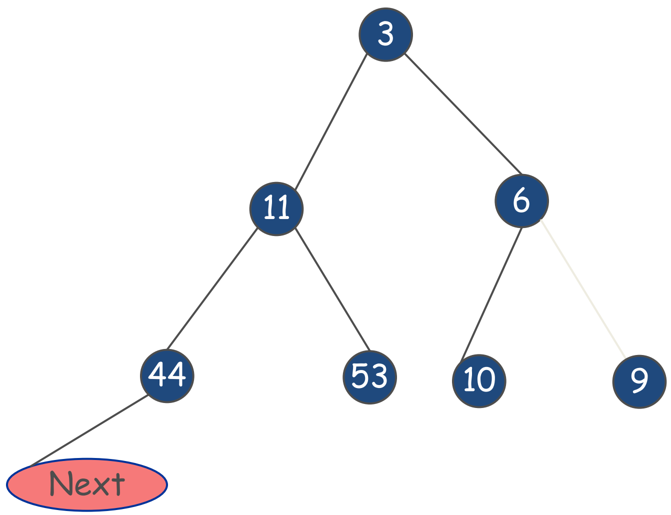


Min Heap Order: For each node v in the tree

$$\text{Parent}(v).\text{Value} \leq v.\text{Value}$$

HEAP INSERTION

Heap.Insert(3)



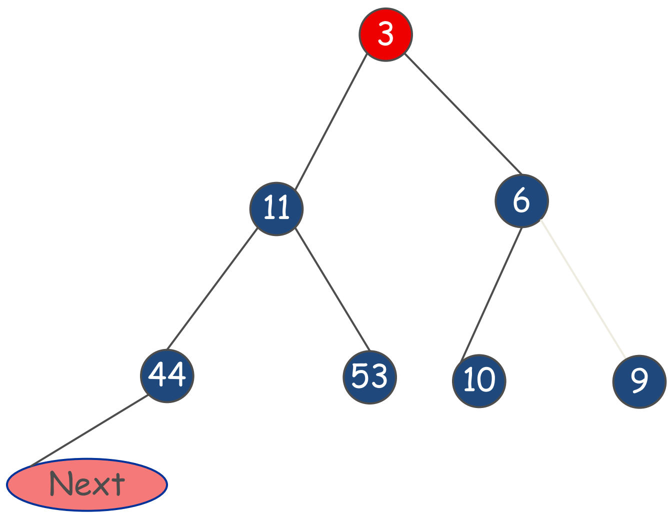
Min Heap Order: For each node v in the tree

$$\text{Parent}(v).\text{Value} \leq v.\text{Value}$$

Theorem 2.12 [KT]: The procedure Heapify-up fixes the heap property and allows us to insert a new element into a heap of n elements in O(log n) time.

HEAP EXTRACT MINIMUM

Heap.ExtractMin()



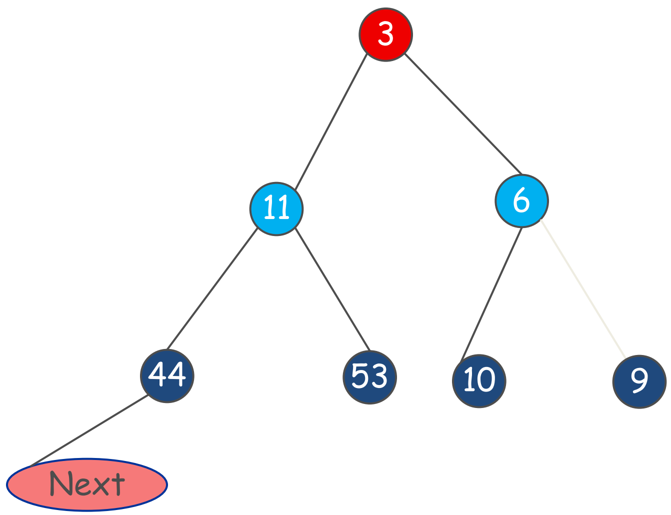
Min Heap Order: For each node v in the tree

$$\text{Parent(v). Value} \leq \text{v. Value}$$

Theorem 2.13 [KT]: The procedure Heapify-down fixes the heap property and allows us to delete an elment in a heap of n elements in O(log n) time.

HEAP EXTRACT MINIMUM

Heap.ExtractMin()



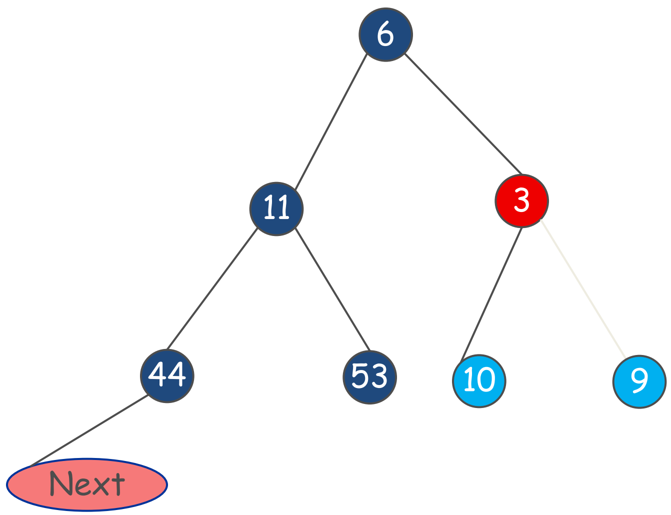
Min Heap Order: For each node v in the tree

$$\text{Parent(v).Value} \leq \text{v.Value}$$

Theorem 2.13 [KT]: The procedure Heapify-down fixes the heap property and allows us to delete an elment in a heap of n elements in O(log n) time.

HEAP EXTRACT MINIMUM

Heap.ExtractMin()



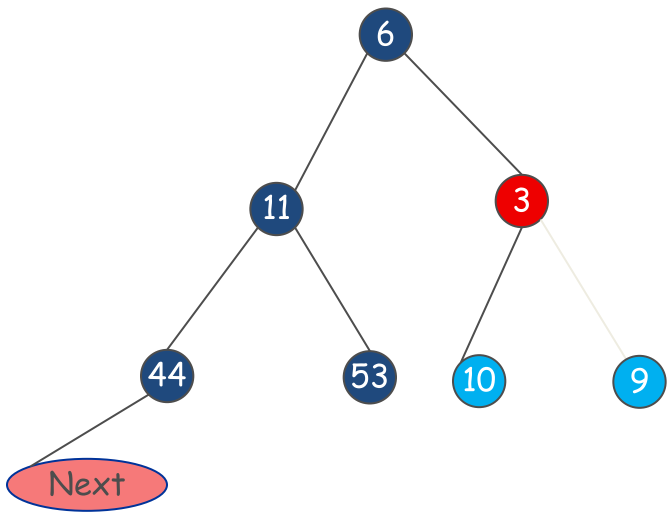
Min Heap Order: For each node v in the tree

$$\mathrm{Parent(v).Value} \leq \mathrm{v.Value}$$

Theorem 2.13 [KT]: The procedure Heapify-down fixes the heap property and allows us to delete an elment in a heap of n elements in O(log n) time.

HEAP EXTRACT MINIMUM

Heap.ExtractMin()



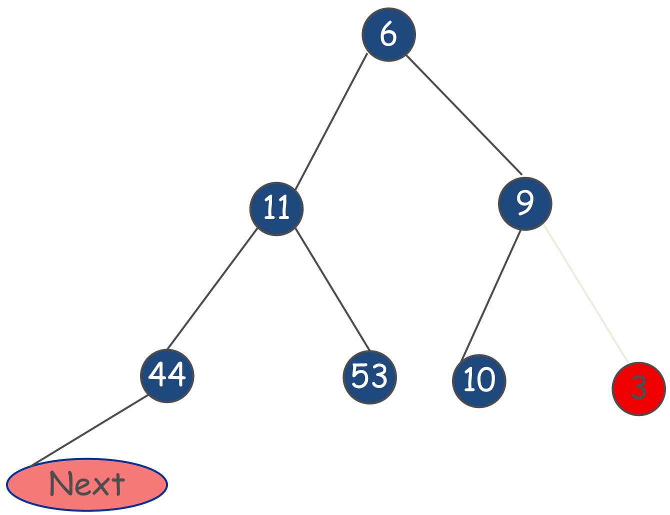
Min Heap Order: For each node v in the tree

$$\text{Parent}(v).\text{Value} \leq v.\text{Value}$$

Theorem 2.13 [KT]: The procedure Heapify-down fixes the heap property and allows us to delete an elment in a heap of n elements in O(log n) time.

HEAP EXTRACT MINIMUM

Heap.ExtractMin()



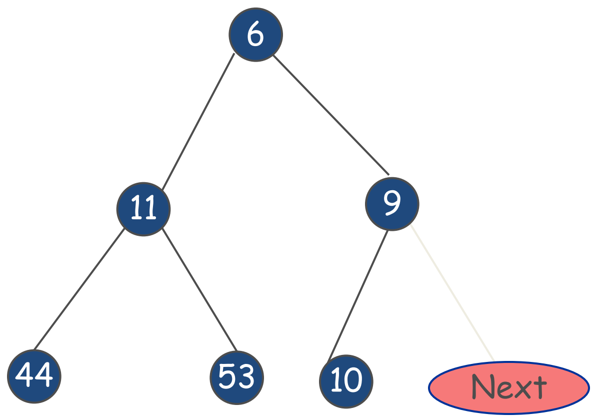
Min Heap Order: For each node v in the tree

$$\text{Parent}(v).\text{Value} \leq v.\text{Value}$$

Theorem 2.13 [KT]: The procedure Heapify-down fixes the heap property and allows us to delete an elment in a heap of n elements in O(log n) time.

HEAP EXTRACT MINIMUM

Heap.ExtractMin()

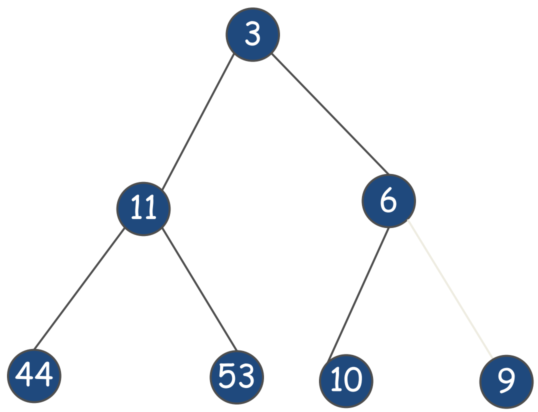


Min Heap Order: For each node v in the tree

$$\text{Parent}(v).\text{Value} \leq v.\text{Value}$$

Theorem 2.13 [KT]: The procedure Heapify-down fixes the heap property and allows us to delete an elment in a heap of n elements in O(log n) time.

HEAP SUMMARY



- Insert: $O(\log n)$
- FindMin: $O(1)$
- Delete(i): $O(\log n)$ time
- ExtractMin: $O(\log n)$ time

SUMMARY

- Why worst case?
- Why polynomial time = good?
- Big-O, Big-Omega, Big-Theta
- Common functions
- Priority queues