Sorting Algorithms

Bin Wang

School of Software
Tsinghua University

March 6, 2018

Outline

Binary Heaps

Definition

The **binary heap** data structure is an array object that can be viewed as a nearly complete binary tree. Each node of the tree corresponds to an element of the array that stores the value in the node.

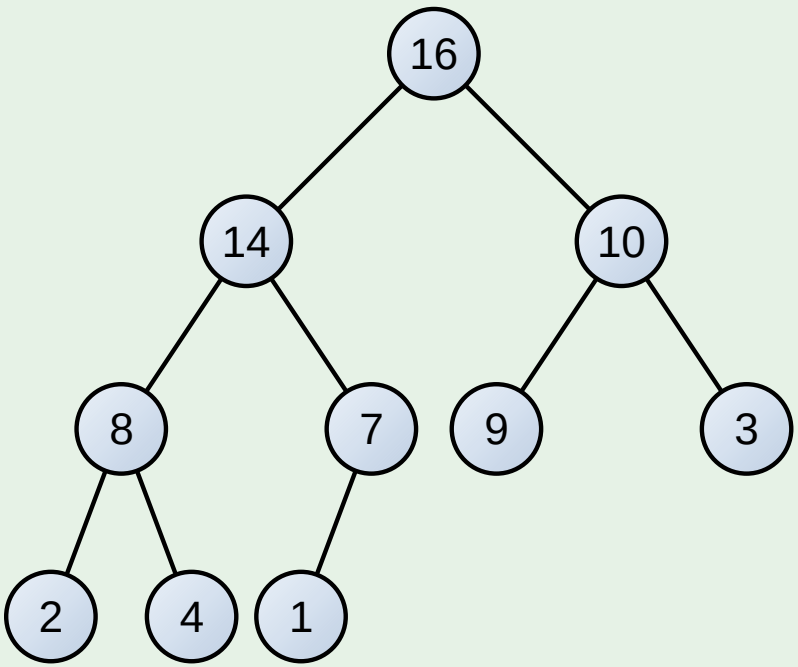
Binary Heaps

Attributes

1. *length*[*A*], which is the number of elements in the array.
2. *heap-size*[*A*], the number of elements in the heap stored within array *A*.

Binary Heaps

Example



16	14	10	8	7	9	3	2	4	1

Binary Heaps

Properties

- PARENT(i): **return** $\lfloor i/2 \rfloor$
- LEFT(i): **return** $2i$
- RIGHT(i): **return** $2i + 1$
- HEIGHT of a n-elements heap: **return** $\lfloor \lg n \rfloor$

Binary Heaps

Properties

- In a **max-heap**, the **max-heap property** is that for every node i other than the root, $A[\text{PARENT}(i)] \geq A[i]$.
- A **min-heap** is organized in the opposite way; the **min-heap property** is that for every node i other than the root, $A[\text{PARENT}(i)] \leq A[i]$.

Maintaining the heap property

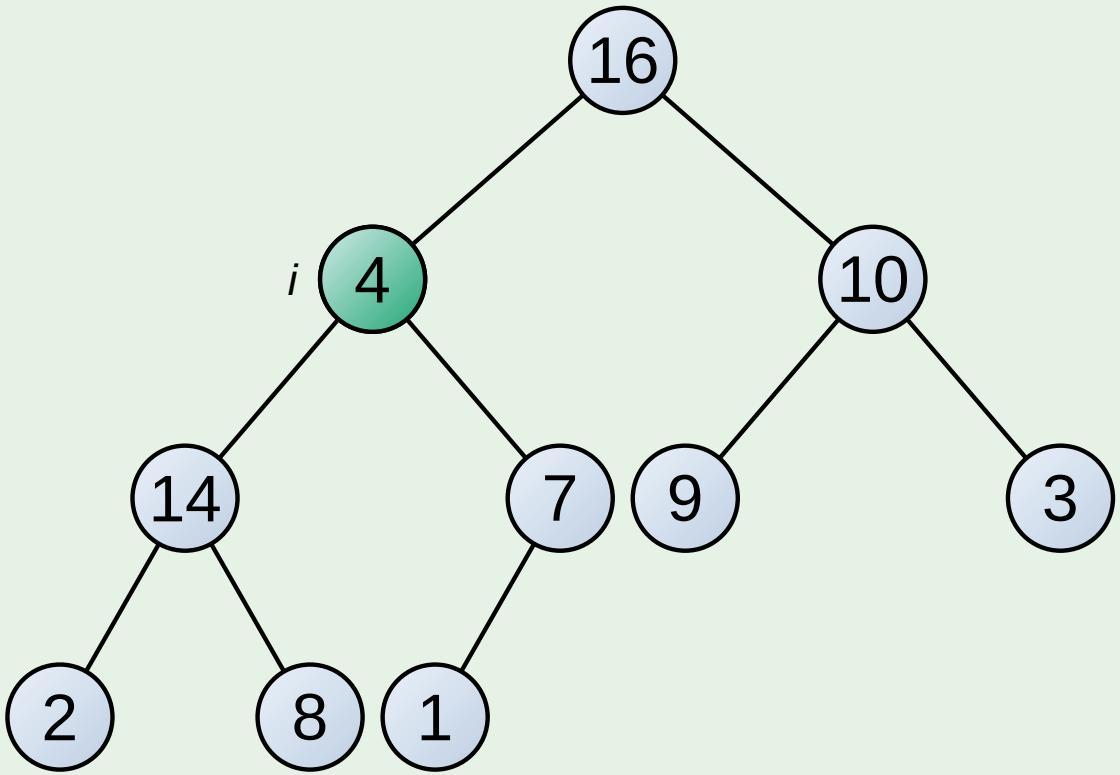
MAX-HEAPIFY(A, i)

```
 1  l = LEFT(i)
 2  r = RIGHT(i)
 3  if l ≤ A.heap-size and A[l] > A[i]
 4      then largest = l
 5      else largest = i
 6  if r ≤ A.heap-size and A[r] > A[largest]
 7      then largest = r
 8  if largest ≠ i
 9      then exchange A[i] ↔ A[largest]
10           MAX-HEAPIFY(A, largest)
```

Skip example

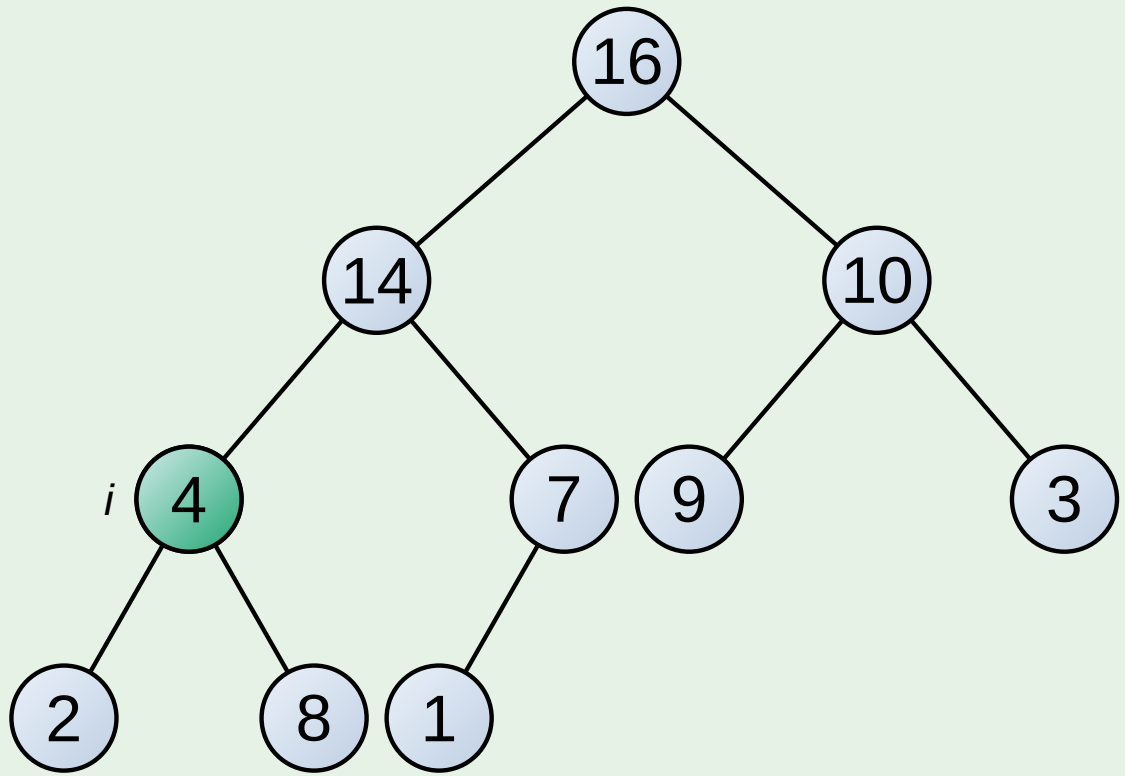
Maintaining the heap property

Example



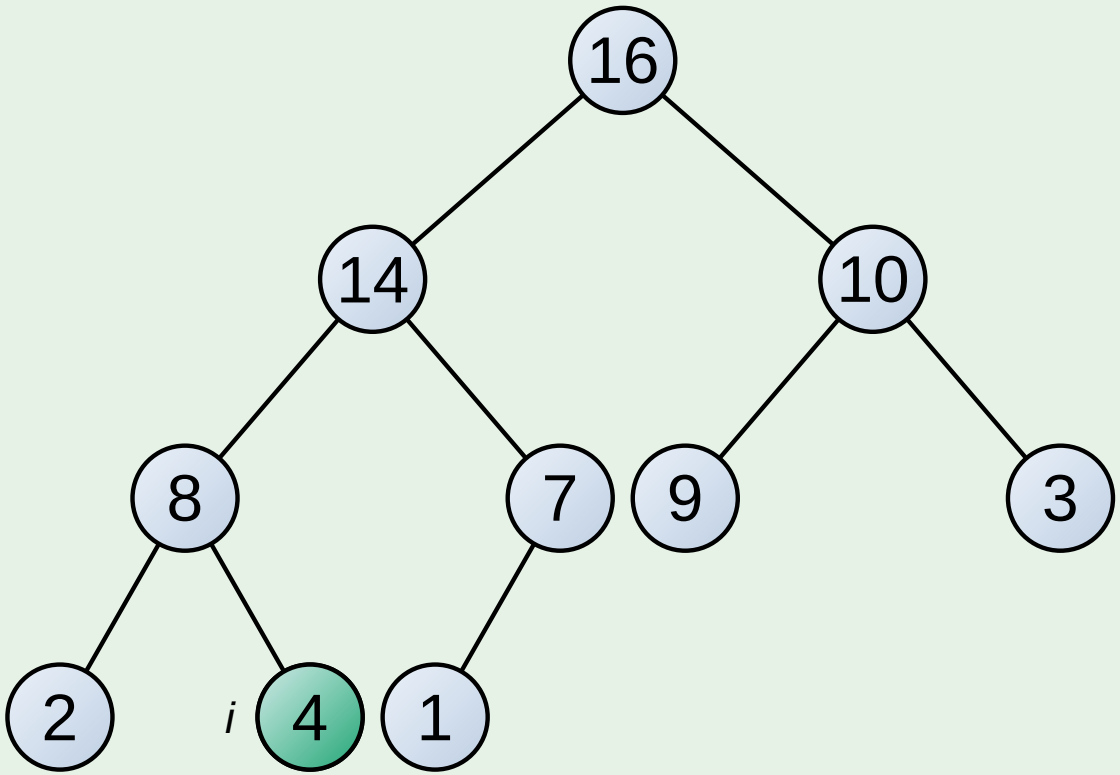
Maintaining the heap property

Example



Maintaining the heap property

Example



Maintaining the heap property

Running time

$$T(n) \leq T(2n/3) + \Theta(1)$$

By Master Theorem, we have $O(\lg n)$

Building a heap

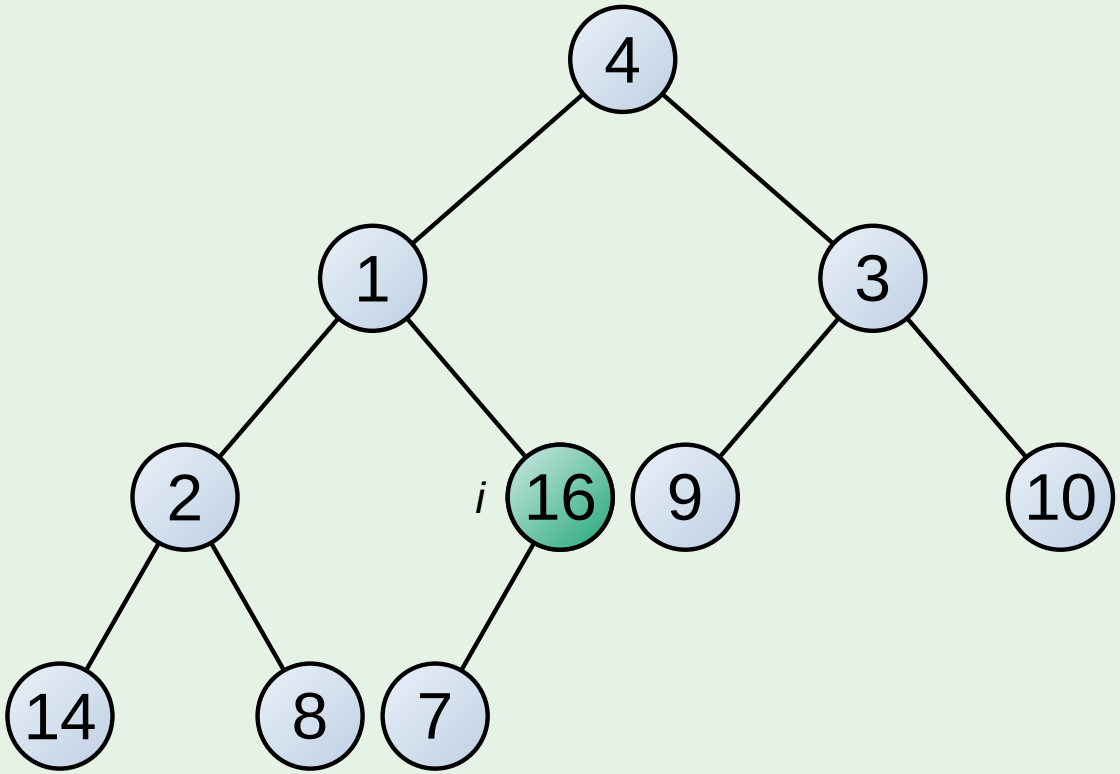
BUILD-MAX-HEAP(A)

1 $A.heap\text{-}size = A.length$
2 **for** $i = \lfloor A.length/2 \rfloor$ **downto** 1
3 **do** MAX-HEAPIFY(A, i)

Skip example

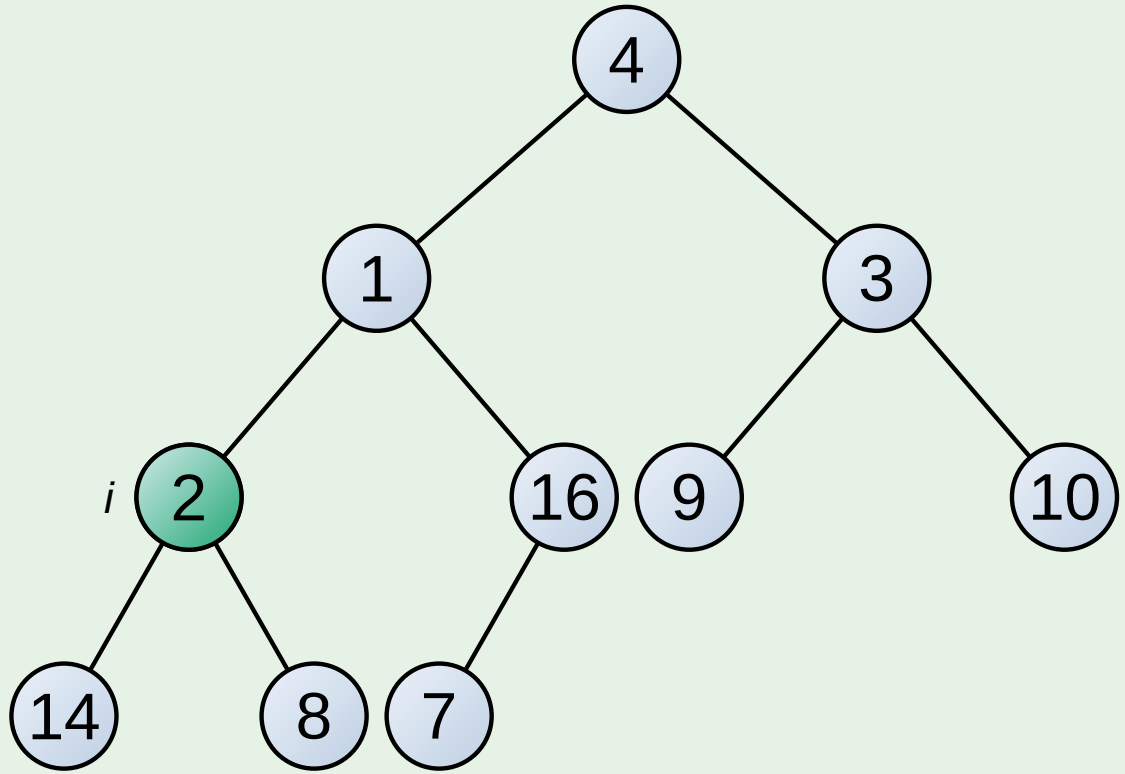
Building a heap

Example



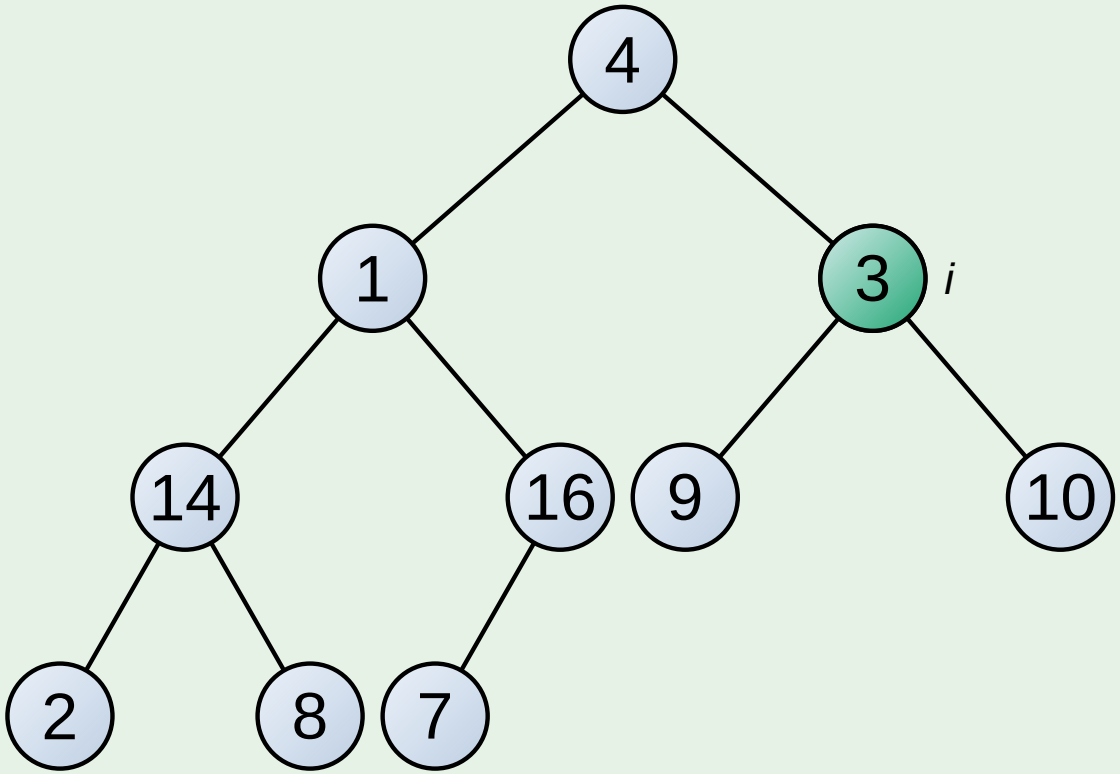
Building a heap

Example



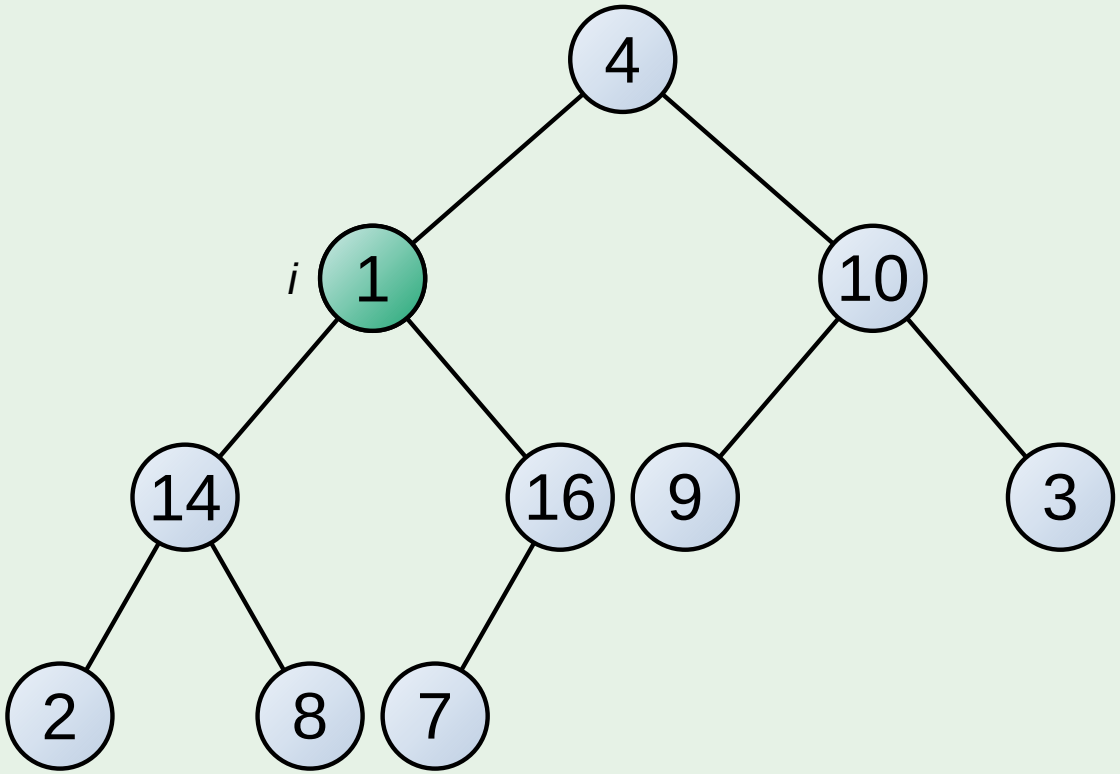
Building a heap

Example



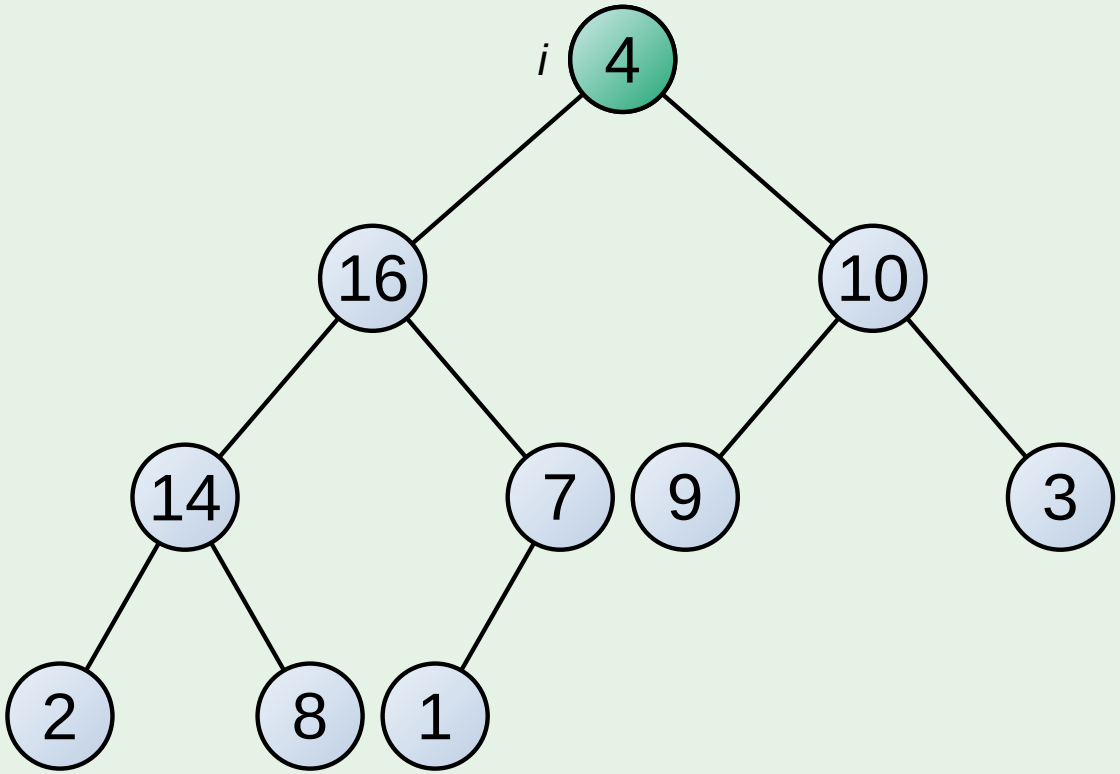
Building a heap

Example



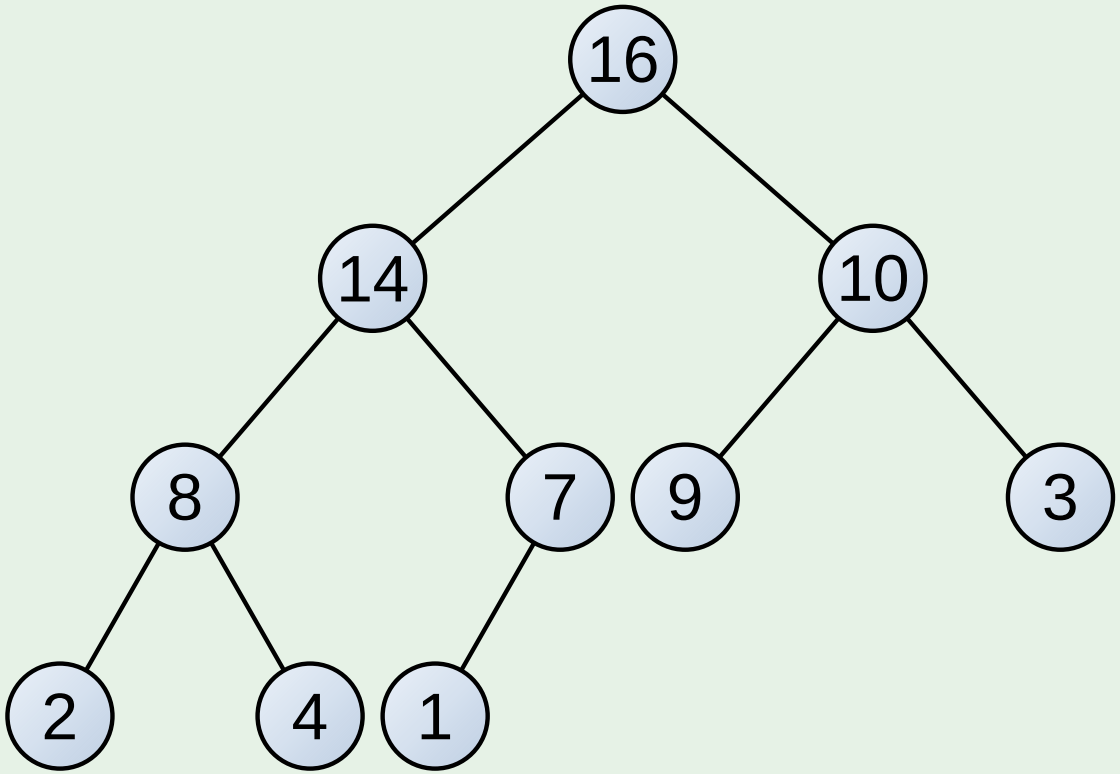
Building a heap

Example



Building a heap

Example



Building a heap

Running time

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h})$$

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

Building a heap

Running time

$$\sum_{h=0}^{\lfloor \lg n \rfloor} \lceil \frac{n}{2^{h+1}} \rceil O(h) = O(n \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h})$$

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

$$= O(n)$$

Heapsort

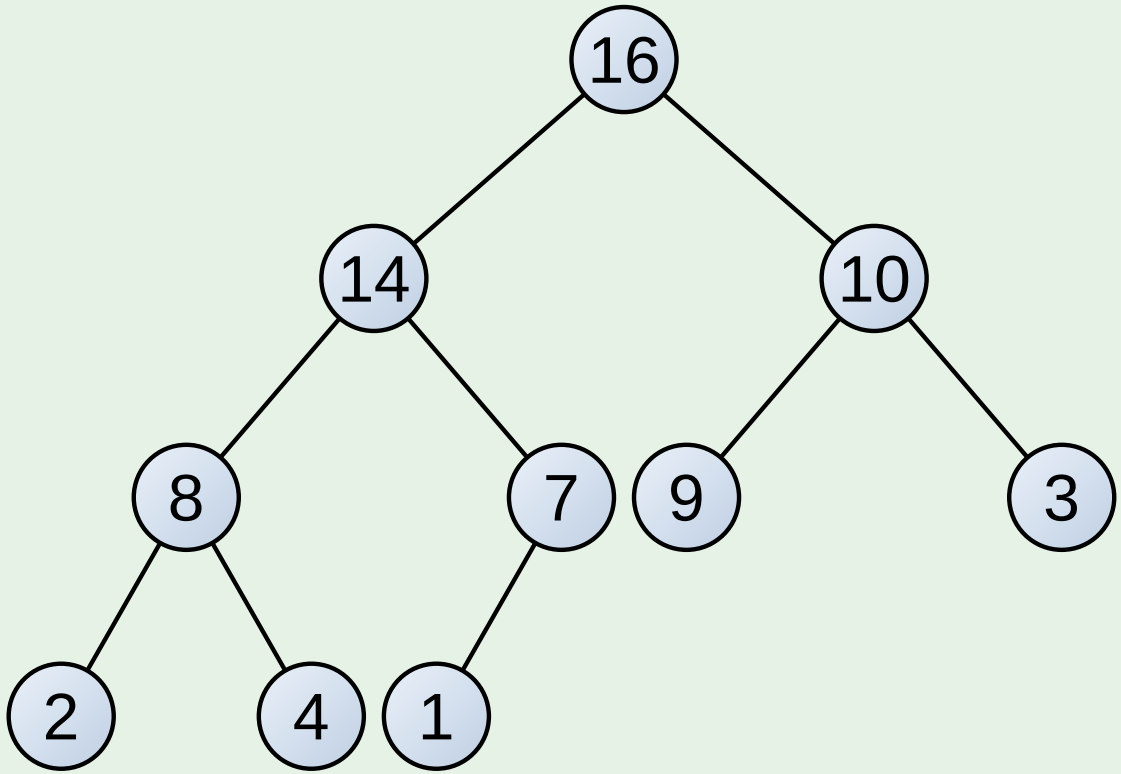
HEAPSORT(A)

```
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3       do exchange A[1] ↔ A[i]
4          A.heap-size = A.heap-size − 1
5          MAX-HEAPIFY(A, 1)
```

Skip example

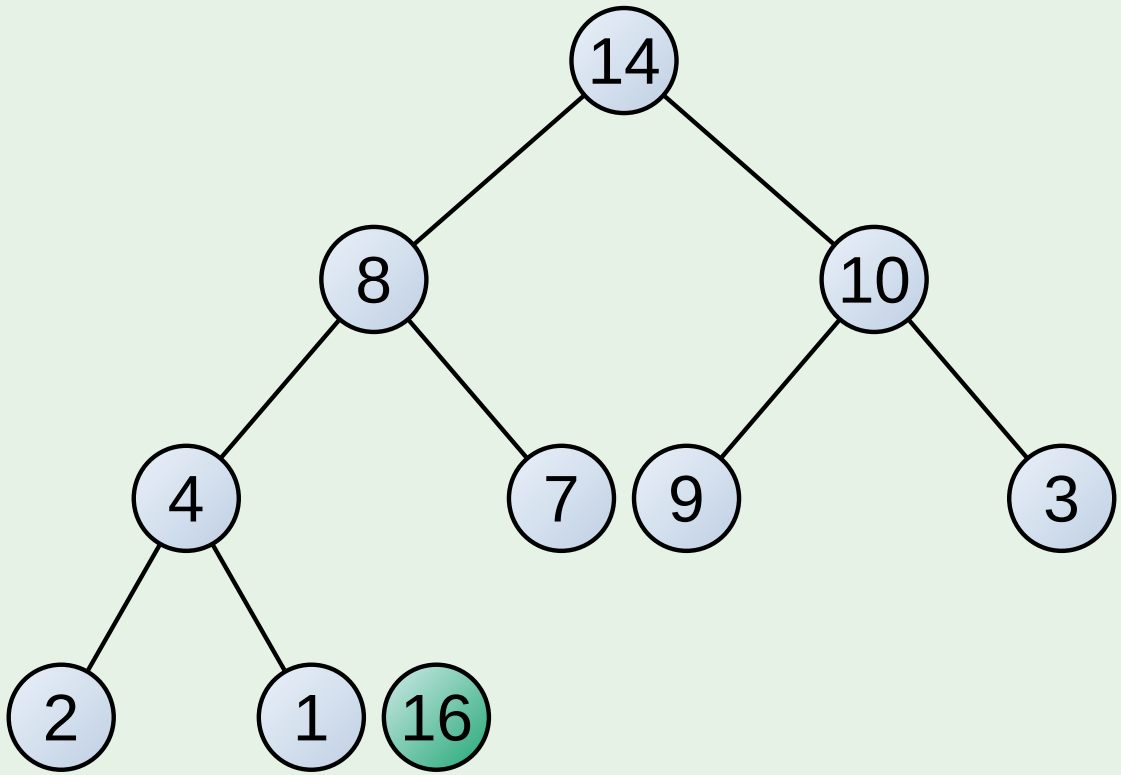
Heapsort

Example



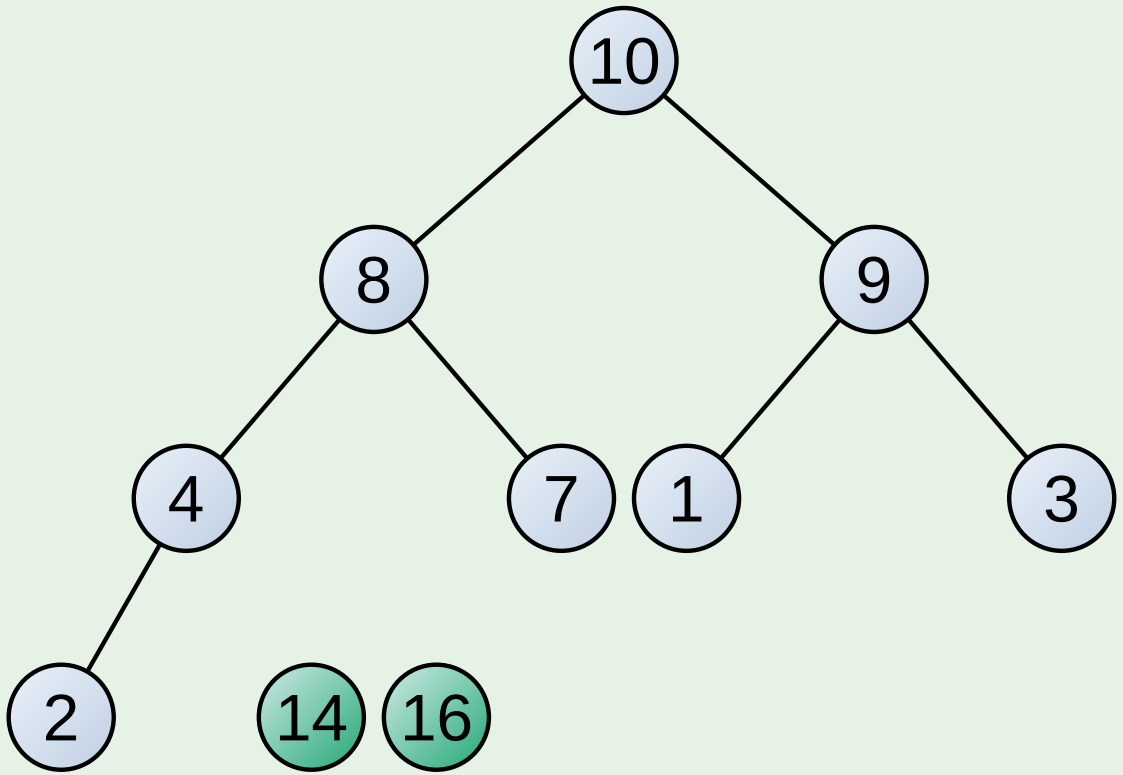
Heapsort

Example



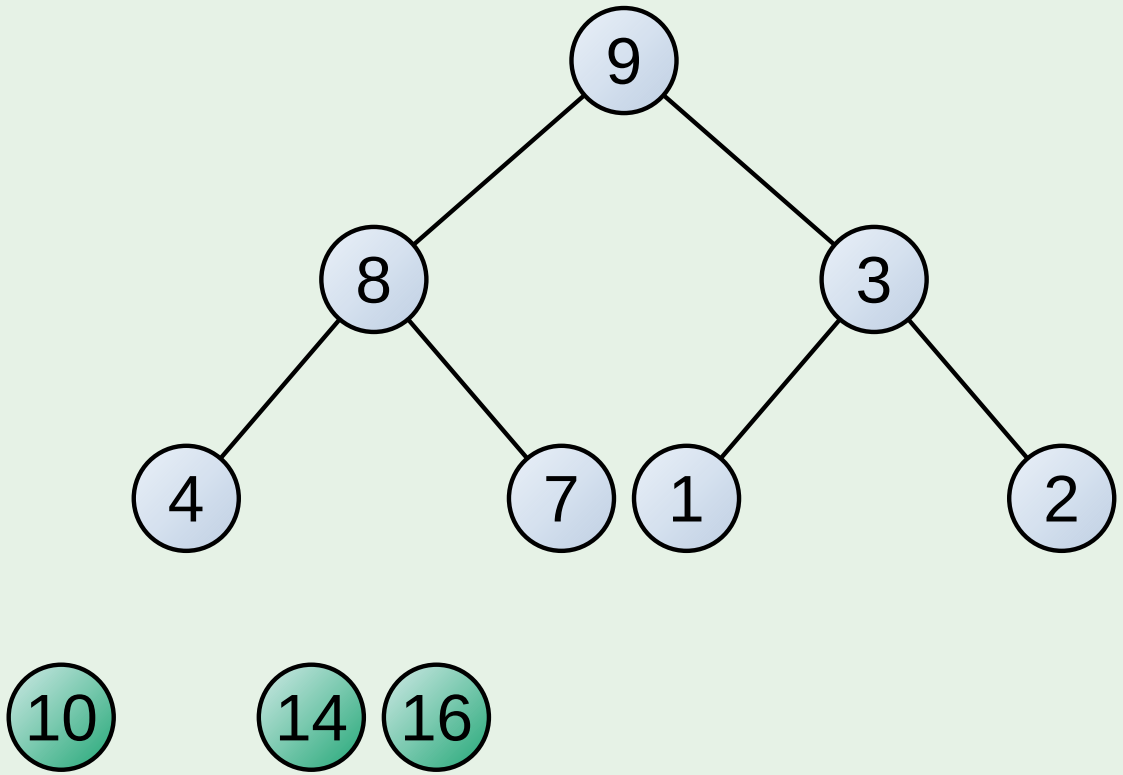
Heapsort

Example



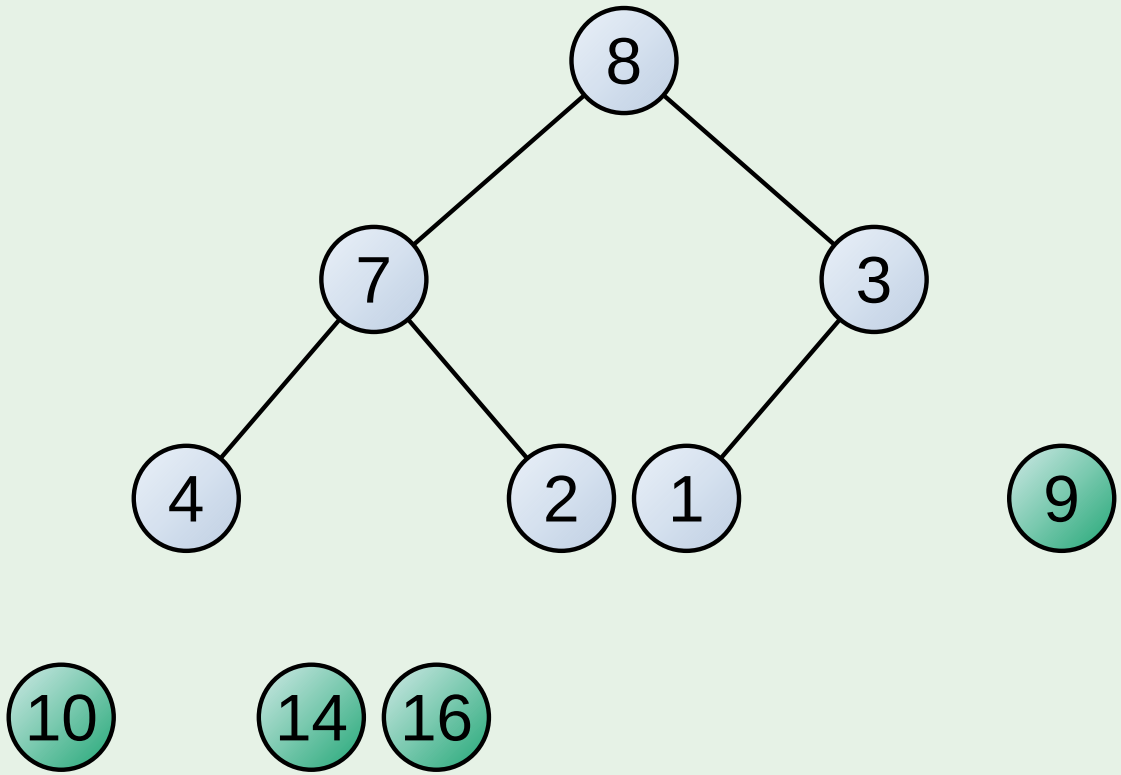
Heapsort

Example



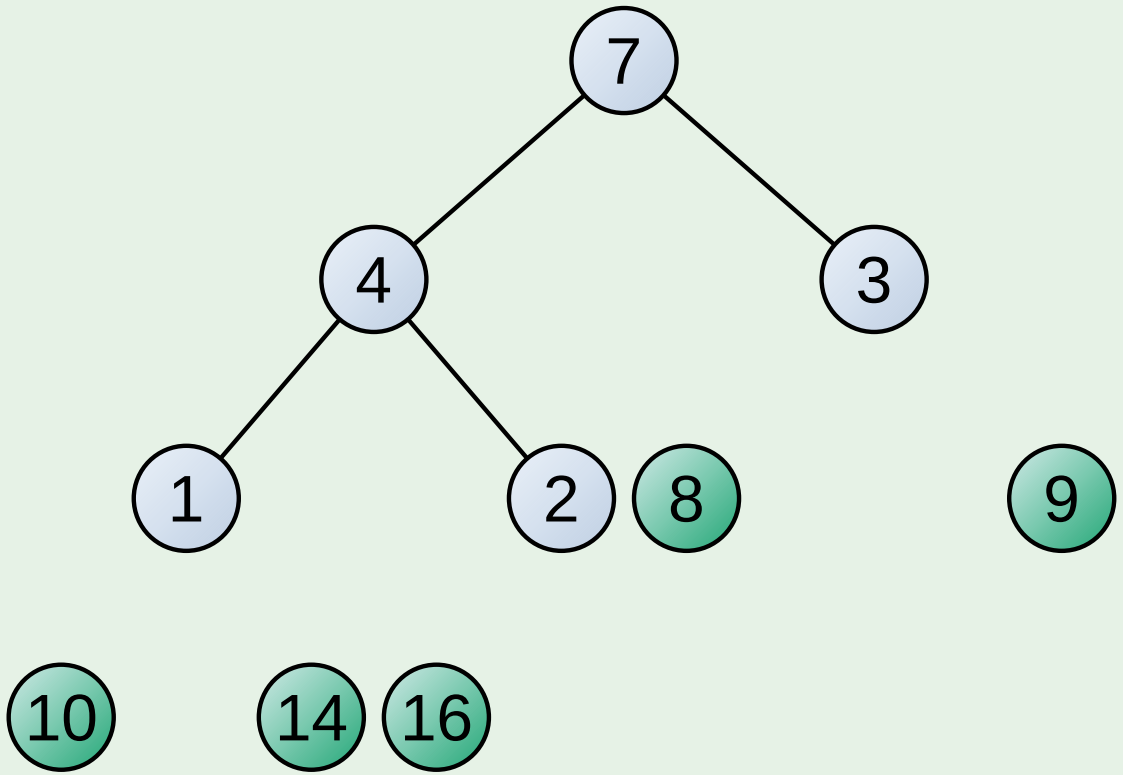
Heapsort

Example



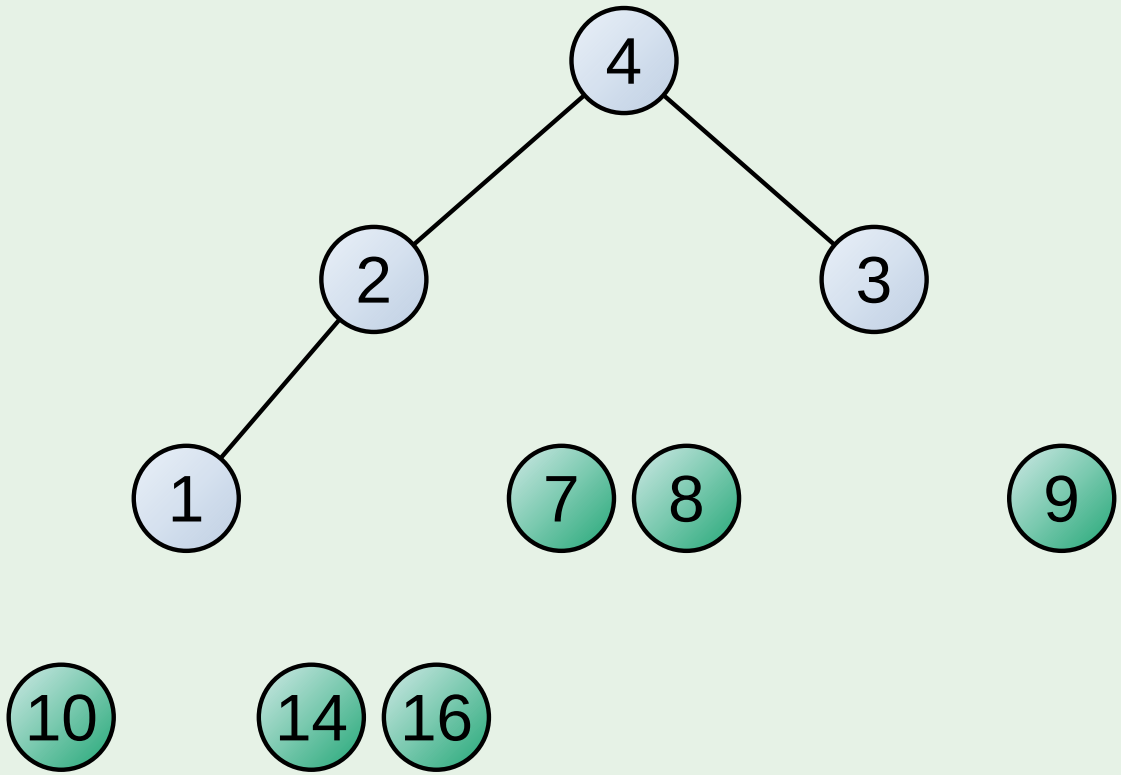
Heapsort

Example



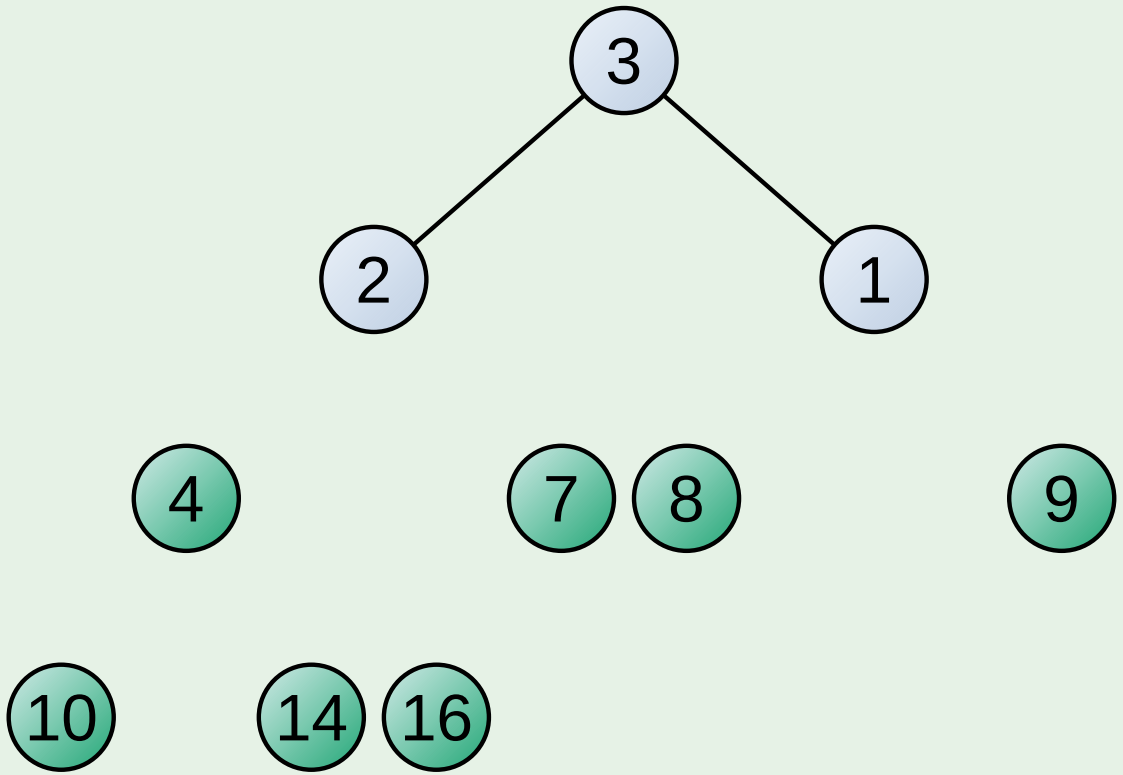
Heapsort

Example



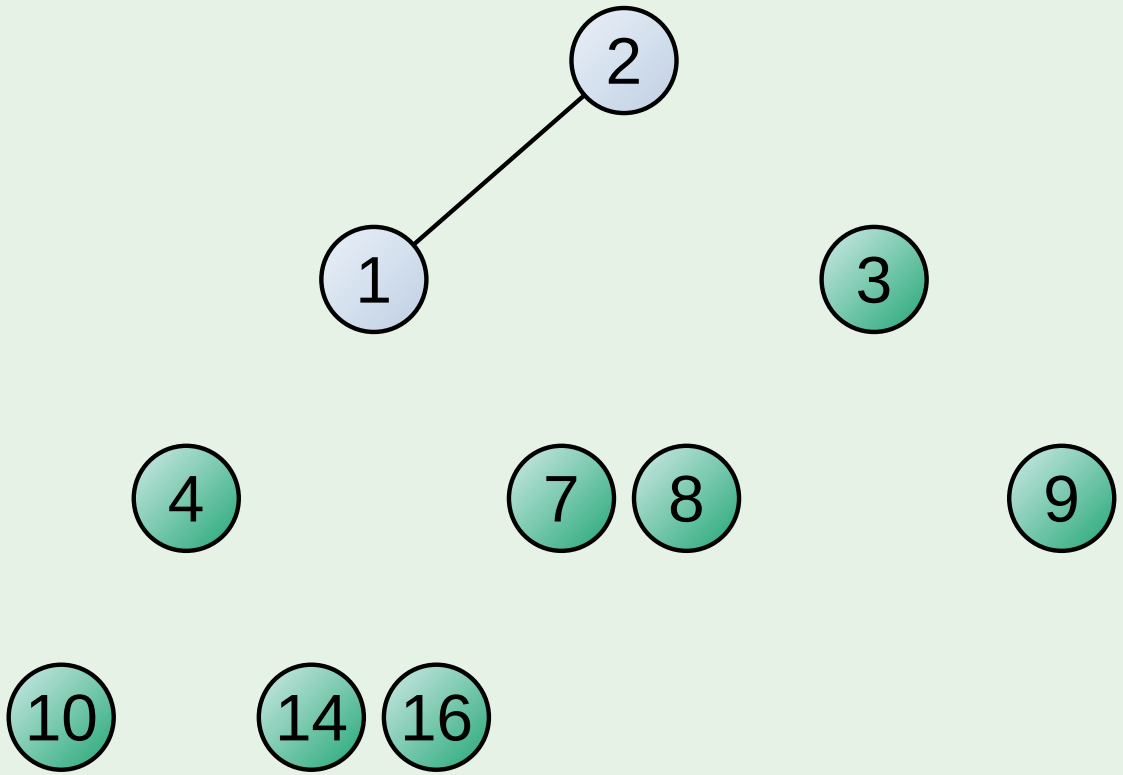
Heapsort

Example



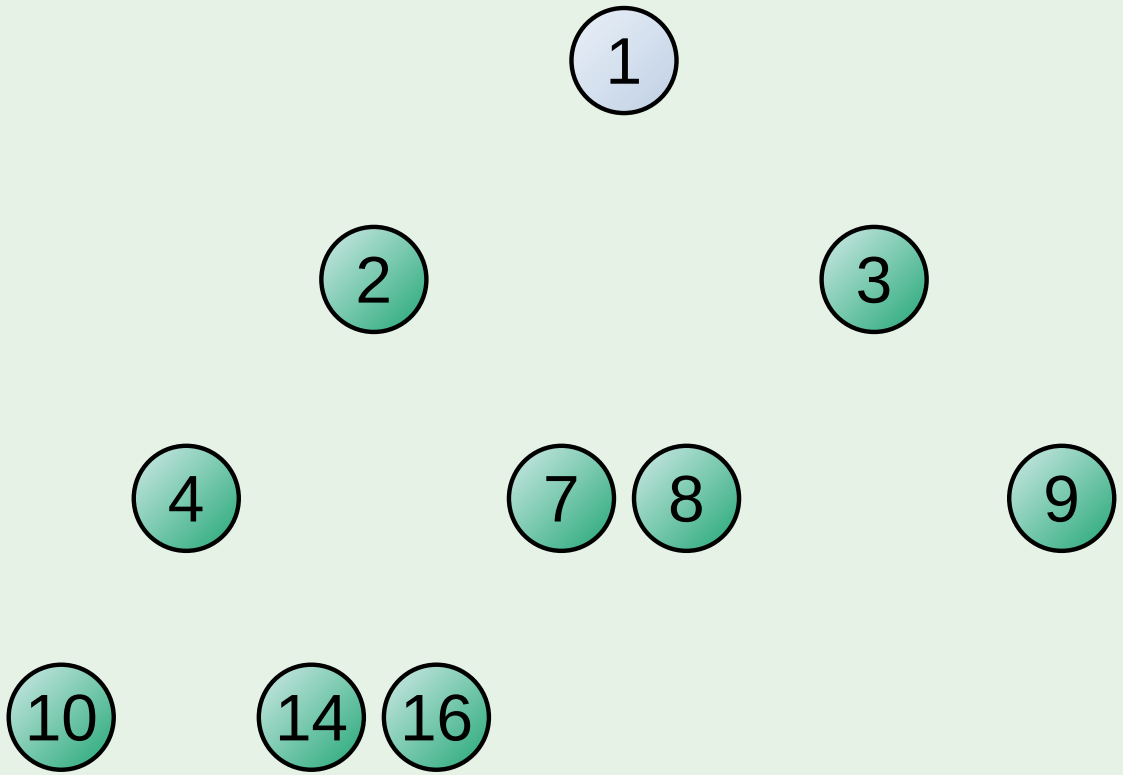
Heapsort

Example



Heapsort

Example



Heapsort

HEAPSORT(A)

```
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3       do exchange A[1] ↔ A[i]
4          A.heap-size = A.heap-size − 1
5          MAX-HEAPIFY(A, 1)
```

Heapsort

HEAPSORT(A)

```
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3       do exchange A[1] ↔ A[i]
4          A.heap-size = A.heap-size − 1
5          MAX-HEAPIFY(A, 1)
```

Running time

$$T(n) \in O(n \lg n)$$

Priority queues

Definition

A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***.

- One application of ***max-priority queues*** is to schedule jobs on a shared computer.
- A ***min-priority queue*** can be used in an event-driven simulator. The items in the queue are events to be simulated, each with an associated time of occurrence that serves as its key.

Priority queues

Definition

A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***.

- We can use a heap to implement a priority queue.
- We often need to store a ***handle*** to the corresponding application object in each heap element.

Priority queues

Definition

A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***. A ***max-priority queue*** supports the following operations.

- INSERT(S, x) inserts the element x into the set S. This operation could be written as $S = S \cup \{x\}$.
- MAXIMUM(S) returns the element of S with the largest key.

Priority queues

Definition

A ***priority queue*** is a data structure for maintaining a set S of elements, each with an associated value called a ***key***. A ***max-priority queue*** supports the following operations.

- INSERT(S, x) inserts the element x into the set S. This operation could be written as $S = S \cup \{x\}$.
- MAXIMUM(S) returns the element of S with the largest key.

Priority queues

Definition

- EXTRACT-MAX(S) removes and returns the element of S with the largest key.
- INCREASE-KEY(S, x, k) increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

Priority queues

Definition

- EXTRACT-MAX(S) removes and returns the element of S with the largest key.
- INCREASE-KEY(S, x, k) increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

Priority queues

HEAP-MAXIMUM(A)

```
1  return A[1]
```

Running time

$\Theta(1)$

HEAP-EXTRACT-MAX(A)

```
1  if heap-size[A] < 1
2      then error "heap underflow
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

Running time

$O(\lg n)$

Priority queues

HEAP-MAXIMUM(A)

```
1  return A[1]
```

Running time

$\Theta(1)$

HEAP-EXTRACT-MAX(A)

```
1  if heap-size[A] < 1
2      then error "heap underflow
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

Running time

$O(\lg n)$

Priority queues

HEAP-MAXIMUM(A)

```
1  return A[1]
```

Running time

$\Theta(1)$

HEAP-EXTRACT-MAX(A)

```
1  if heap-size[A] < 1
2      then error "heap underflow
3  max = A[1]
4  A[1] = A[A.heap-size]
5  A.heap-size = A.heap-size − 1
6  MAX-HEAPIFY(A, 1)
7  return max
```

Running time

$O(\lg n)$

Priority queues

HEAP-INCREASE-KEY(A, i, key)

```
1  if key < A[i]
2      then error "new key is smaller"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5        do exchange A[i] ↔ A[PARENT(i)]
6           i = PARENT(i)
```

Running time

$O(\lg n)$

Skip example

Priority queues

HEAP-INCREASE-KEY(A, i, key)

```
1  if key < A[i]
2      then error "new key is smaller"
3  A[i] = key
4  while i > 1 and A[PARENT(i)] < A[i]
5        do exchange A[i] ↔ A[PARENT(i)]
6           i = PARENT(i)
```

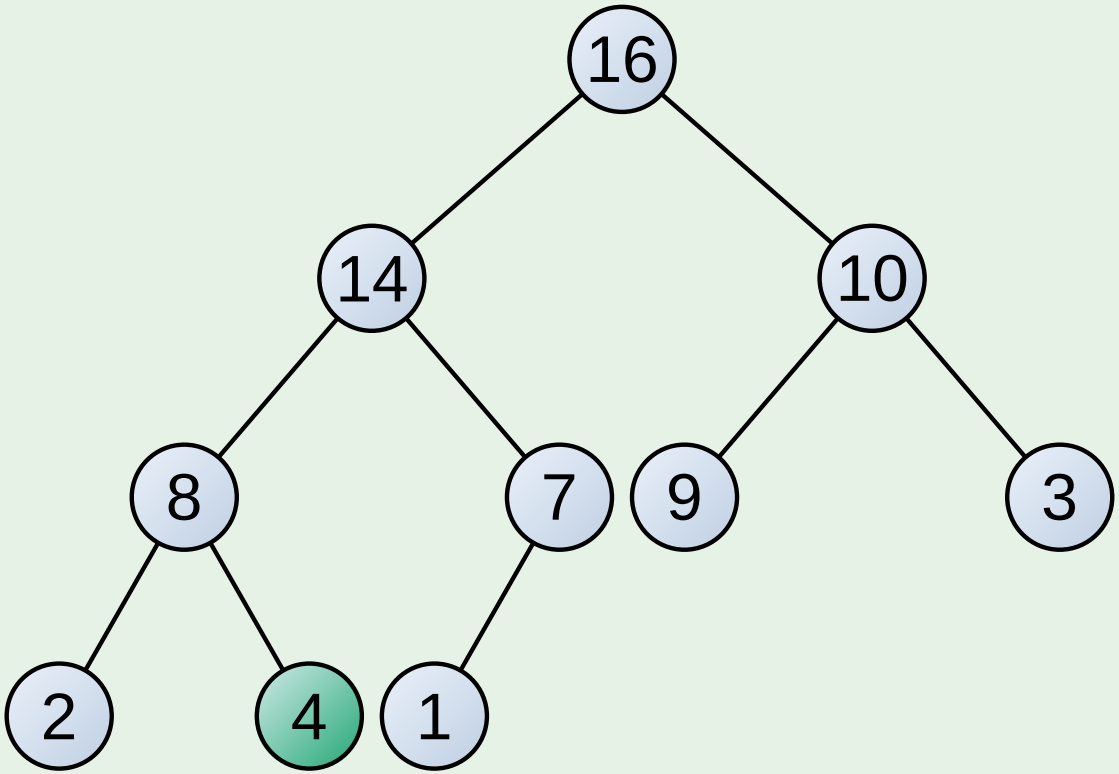
Running time

$$O(\lg n)$$

Skip example

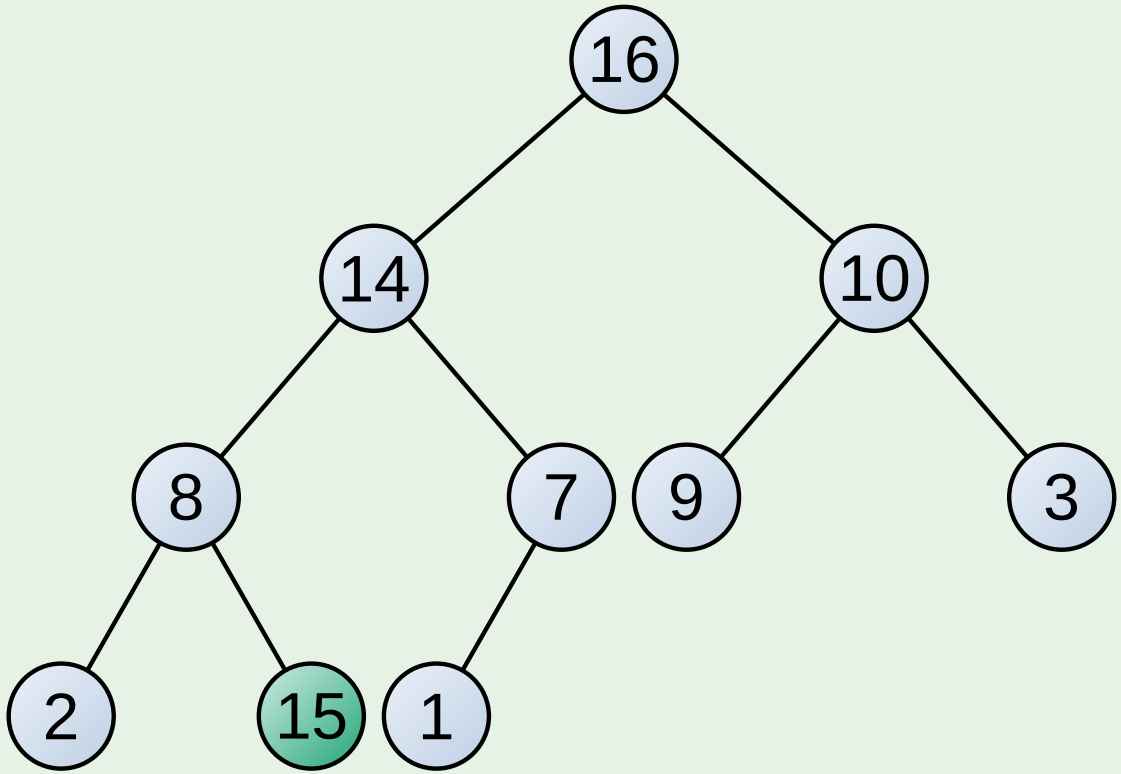
Priority queues

Example



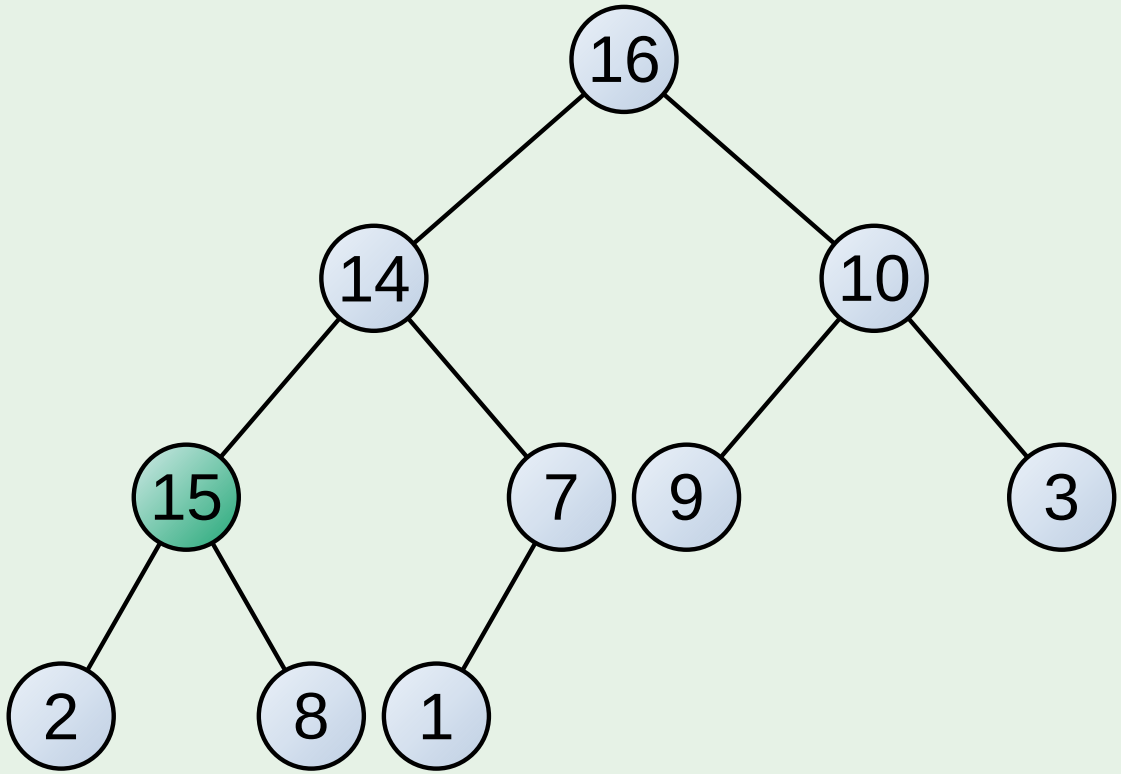
Priority queues

Example



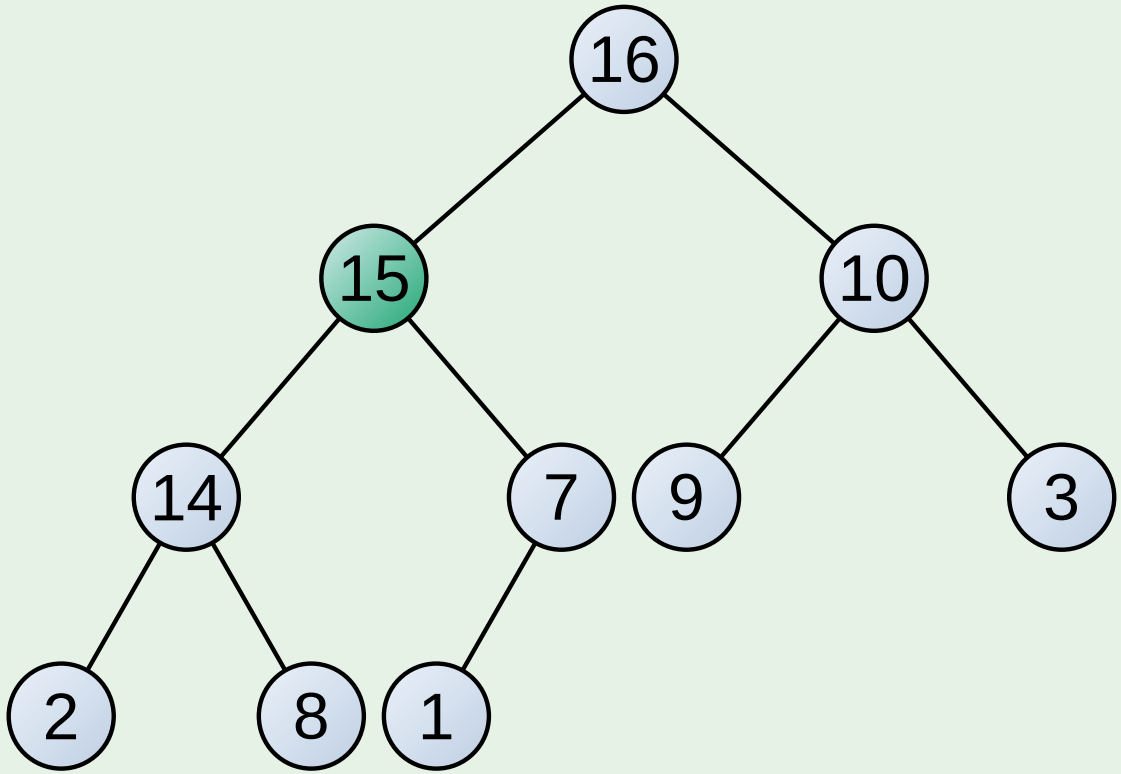
Priority queues

Example



Priority queues

Example



Priority queues

MAX-HEAP-INSERT(A, key)

1 $A.heap\text{-}size = A.heap\text{-}size + 1$
2 $A[A.heap\text{-}size] = -\infty$
3 HEAP-INCREASE-KEY($A, heap\text{-}size[A], key$)

Running time

$O(\lg n)$

Priority queues

MAX-HEAP-INSERT(A, key)

1 $A.heap\text{-}size = A.heap\text{-}size + 1$
2 $A[A.heap\text{-}size] = -\infty$
3 HEAP-INCREASE-KEY($A, heap\text{-}size[A], key$)

Running time

$O(\lg n)$

History

- The heapsort algorithm was invented by J.W.J. Williams in 1964. The BUILD-MAX-HEAP procedure was suggested by R.W. Floyd in 1964.
- If the data are b-bit integers and the computer memory consists of addressable $b-$bit words, Fredman and Willard showed how to implement MINIMUM in $O(1)$ time and INSERT and EXTRACT-MIN in $O(\sqrt{\lg n})$ in 1993.

History

- In 2000, M. Thorup improved the $O(\sqrt{\lg n})$ bound to $O(\lg \lg n)$ time. This bound uses an amount of space unbounded in n, but it can be implemented in linear space by using randomized hashing.

Quicksort

Overview

- The quicksort procedure was invented by C.A.R. Hoare in 1962.
- Use the divide-and-conquer idea.
- It is an “in place” sorting algorithm.
- It is very practical and widely used.

Idea

Divide and Conquer

Quicksort an n-element array:

- **Divide:** Partition the array into two subarrays around a **pivot** x such that *elements in lower subarray* $\leq x <$ *elements in upper subarray*.
- **Conquer:** Recursively sort the two subarrays.
- **Combine:** Trivial.

Key: Partitioning subroutine.

Idea

Divide and Conquer

Quicksort an n-element array:

- **Divide:** Partition the array into two subarrays around a **pivot** x such that *elements in lower subarray* $\leq x <$ *elements in upper subarray*.
- **Conquer:** Recursively sort the two subarrays.
- **Combine:** Trivial.

  Key: Partitioning subroutine.

Partitioning the array

PARTITION(A, p, r)

```
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4        do if A[j] ≤ x
5              then i = i + 1
6                   exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

Running time $= O(n)$ for n elements.

Partitioning the array

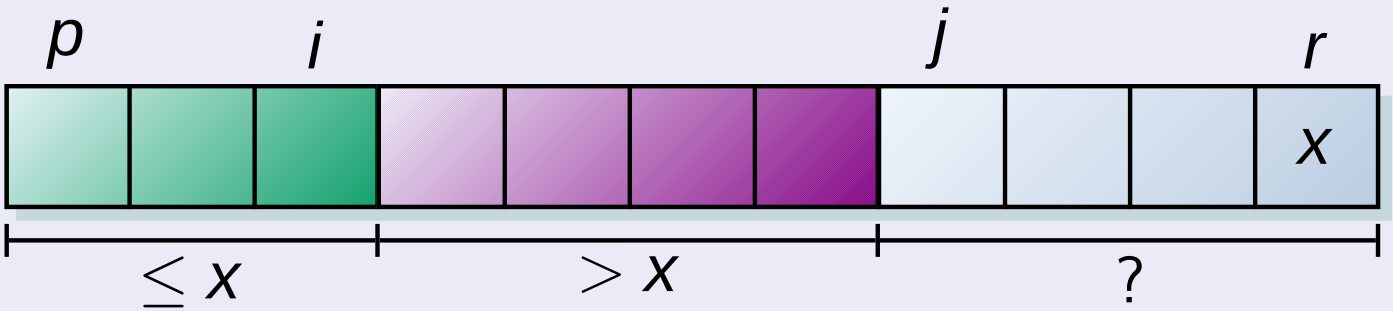
PARTITION(A, p, r)

```
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4        do if A[j] ≤ x
5              then i = i + 1
6                   exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return i + 1
```

Running time $= O(n)$ for n elements.

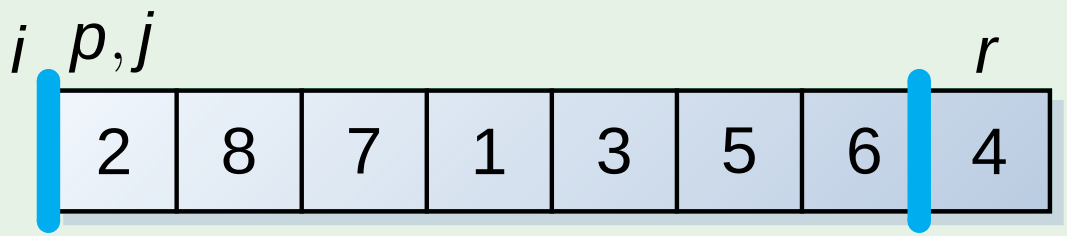
Partitioning the array

Invariant:



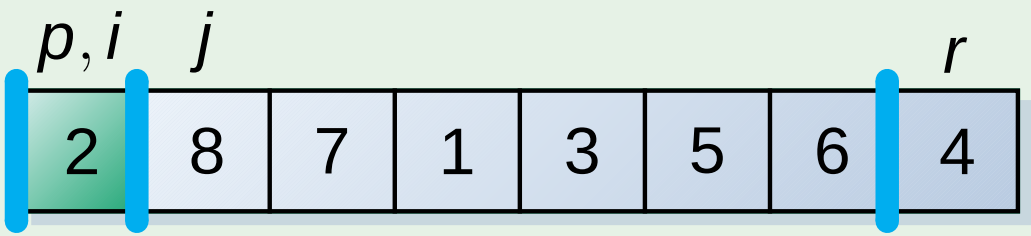
Partitioning the array

Example



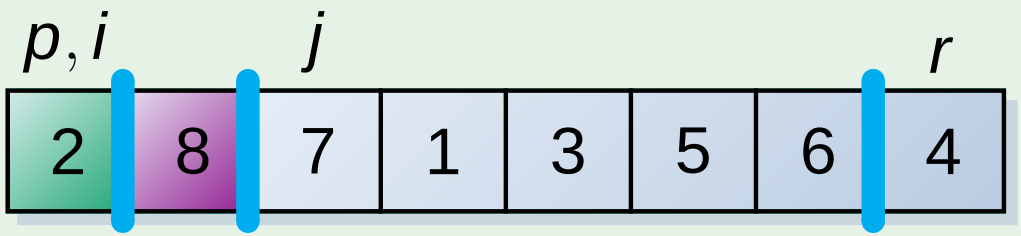
Partitioning the array

Example



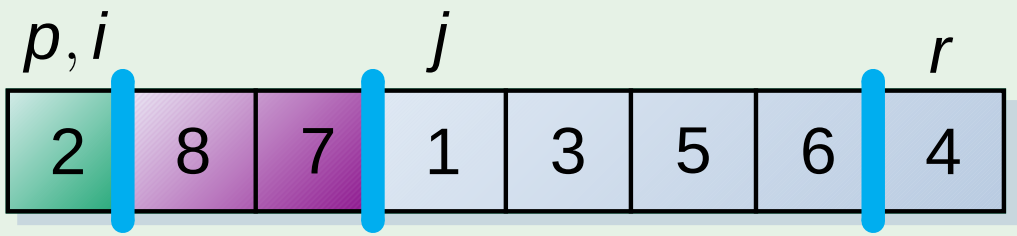
Partitioning the array

Example



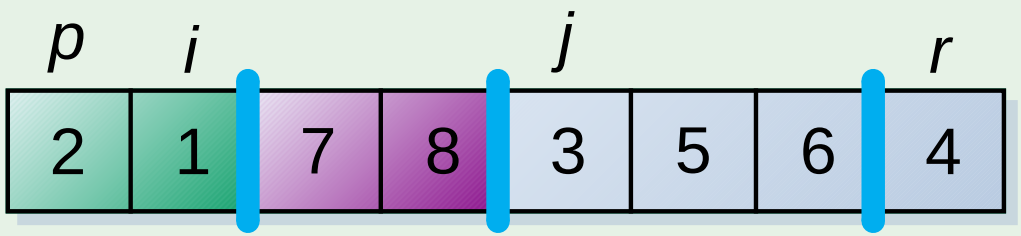
Partitioning the array

Example



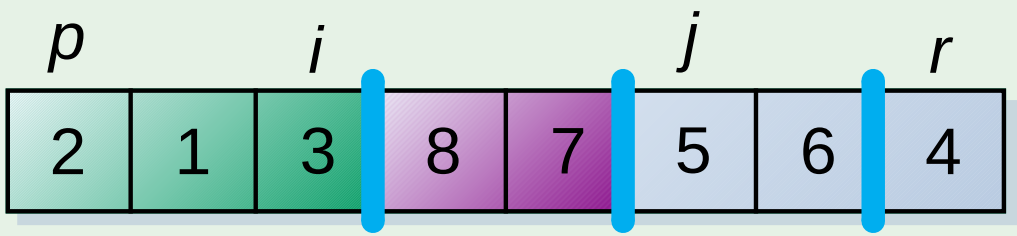
Partitioning the array

Example



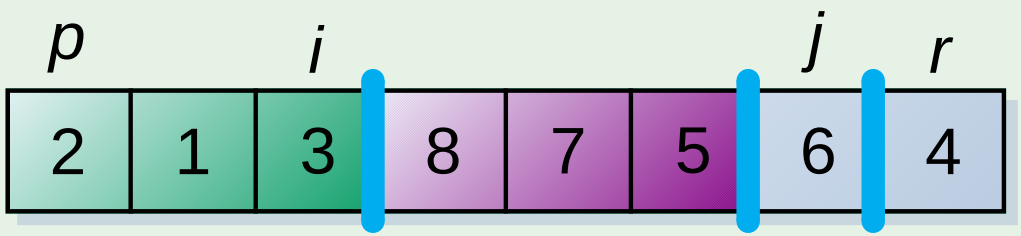
Partitioning the array

Example



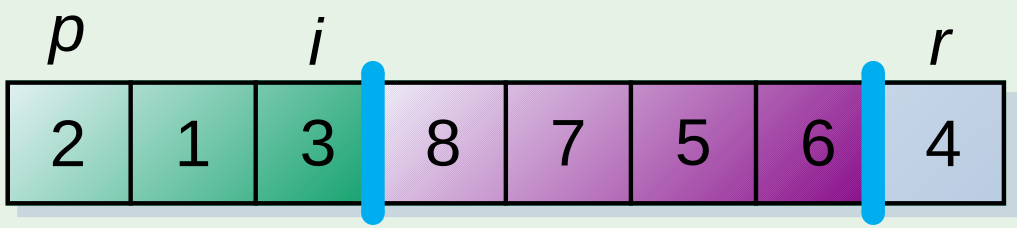
Partitioning the array

Example



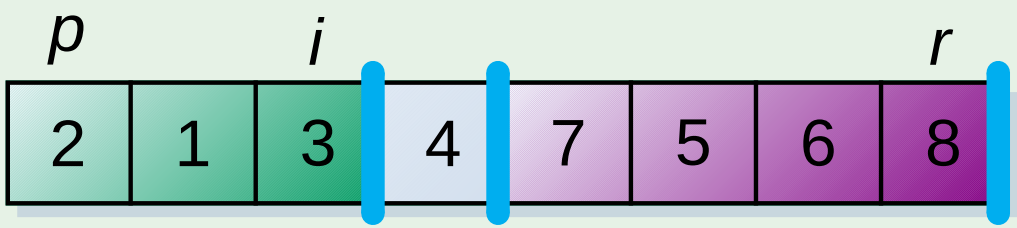
Partitioning the array

Example



Partitioning the array

Example



Pseudocode for quicksort

```
QUICKSORT(A, p, r)
1  if p < r
2     then q = PARTITION(A, p, r)
3          QUICKSORT(A, p, q − 1)
4          QUICKSORT(A, q + 1, r)
```

Initial call: QUICKSORT$(A, 1, n)$

Pseudocode for quicksort

QUICKSORT(A, p, r)

```
1   if p < r
2       then q = PARTITION(A, p, r)
3            QUICKSORT(A, p, q − 1)
4            QUICKSORT(A, q + 1, r)
```

Initial call: QUICKSORT$(A, 1, n)$

Analysis of quicksort

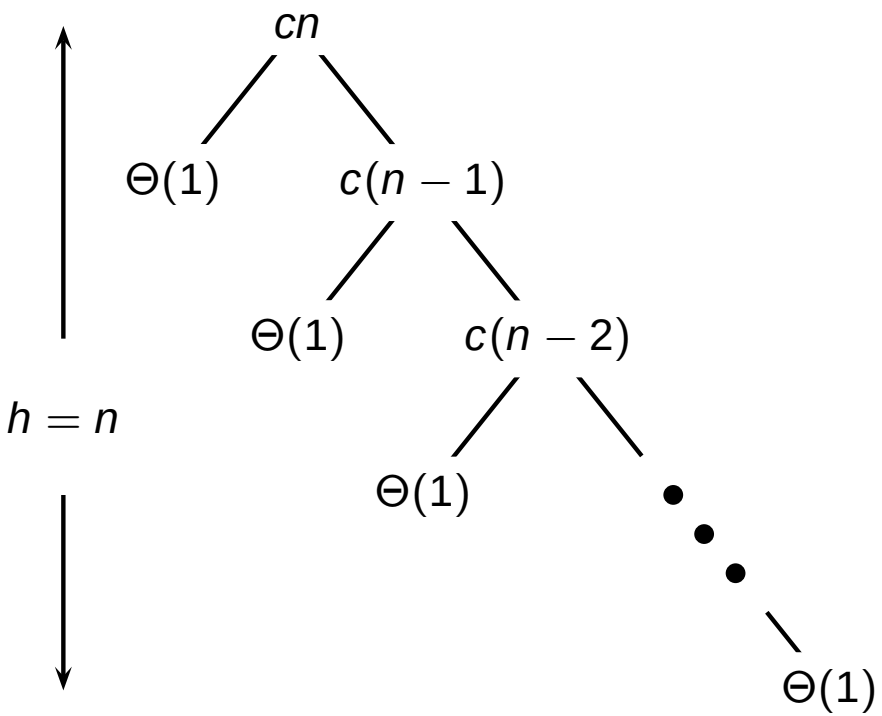
Worst-case of quicksort

- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.

$$\begin{aligned} T(n) &= T(0) + T(n-1) + \Theta(n) \\ &= \Theta(1) + T(n-1) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ &= \Theta(n^2) \end{aligned}$$

Worst-case recursion tree

$$T(n) = T(0) + T(n-1) + \Theta(n)$$



$$\Theta(\sum_{k=1}^{n} k) = \Theta(n^2)$$

$$T(n) = \Theta(n) + \Theta(n^2)$$
$$= \Theta(n^2)$$

Best-case analysis

- In the most even possible split, PARTITION produces two subproblem, each of size no more than $n/2$.
- The recurrence for the running time is then

$$\begin{aligned} T(n) &= 2T(n/2) + \Theta(n) \\ &= \Theta(n \lg n) \end{aligned}$$

Best-case analysis

- What if the split is always $\frac{1}{10} : \frac{9}{10}$?

$$T(n) = T(\frac{1}{10}n) + T(\frac{9}{10}n) + \Theta(n)$$

What is the solution to this recurrence?

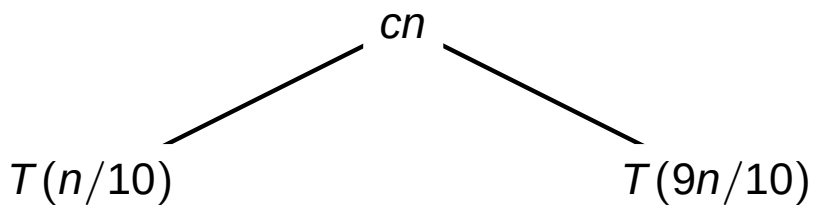
Analysis of “almost-best” case

$T(n)$

$$cn \lg_{10} n \leq T(n) \leq cn \lg_{10/9} n$$

$$T(n) \in \Theta(n \lg n)$$

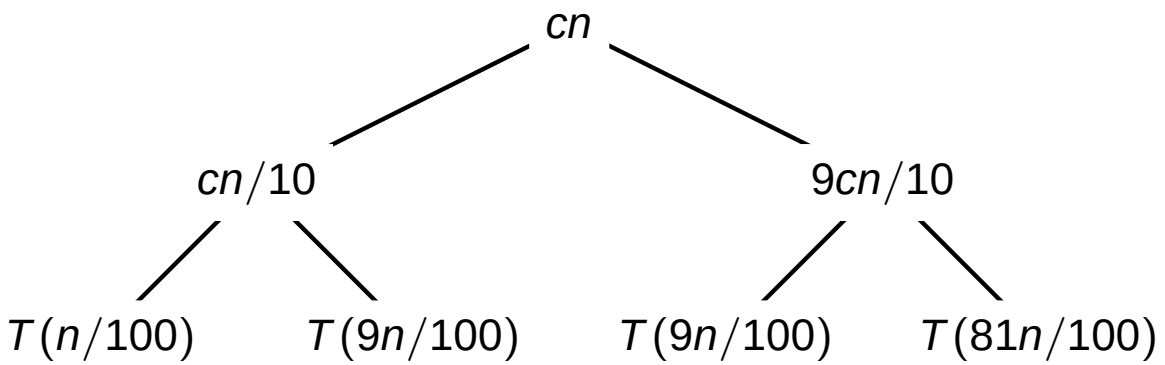
Analysis of “almost-best” case



$$cn \lg_{10} n \leq T(n) \leq cn \lg_{10/9} n$$

$$T(n) \in \Theta(n \lg n)$$

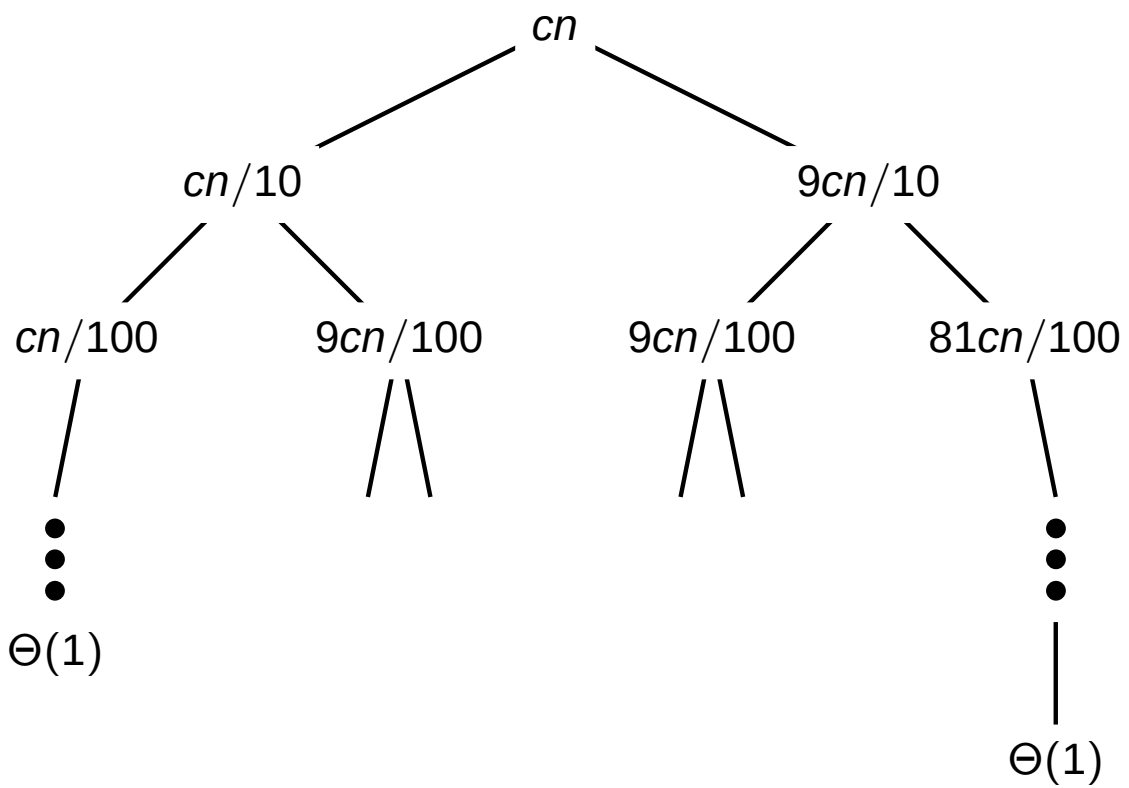
Analysis of “almost-best” case



$$cn \lg_{10} n \leq T(n) \leq cn \lg_{10/9} n$$

$$T(n) \in \Theta(n \lg n)$$

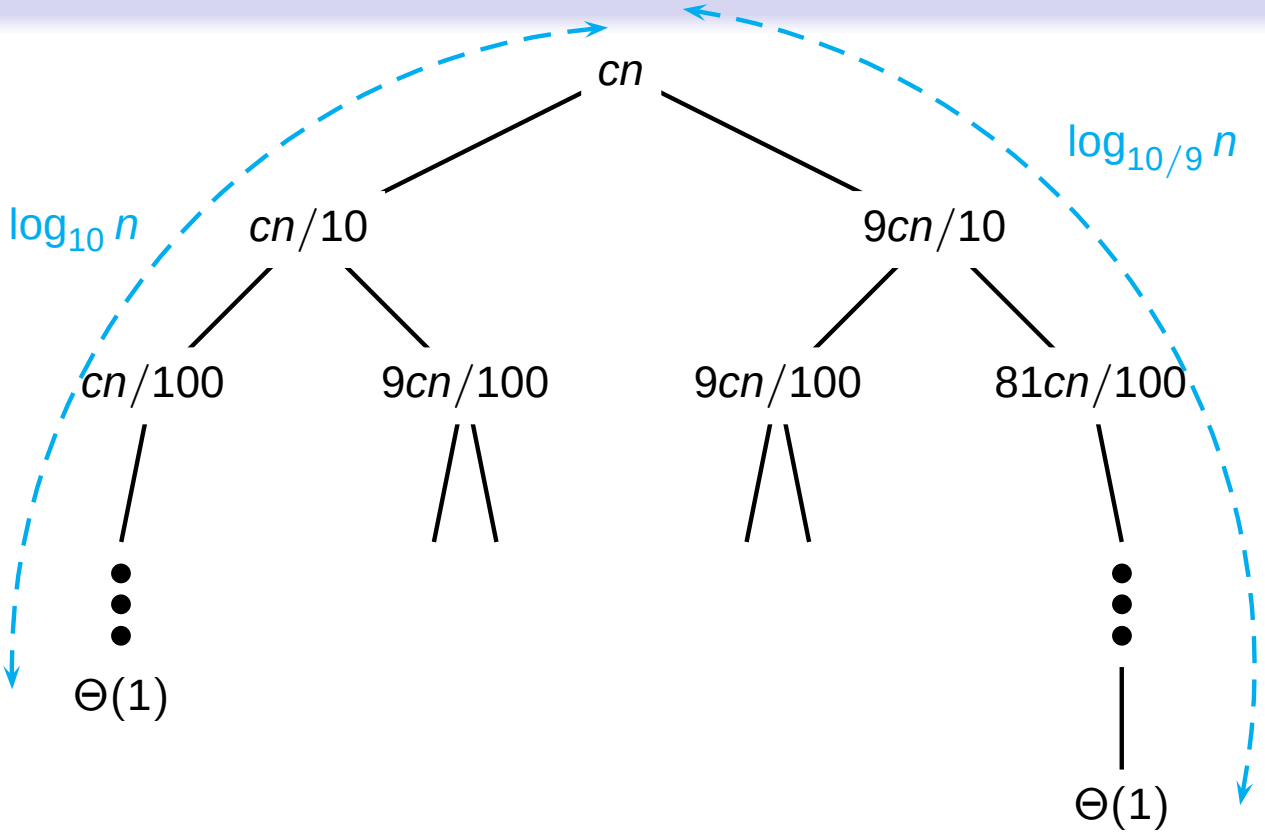
Analysis of “almost-best” case



$$cn \lg_{10} n \leq T(n) \leq cn \lg_{10/9} n$$

$$T(n) \in \Theta(n \lg n)$$

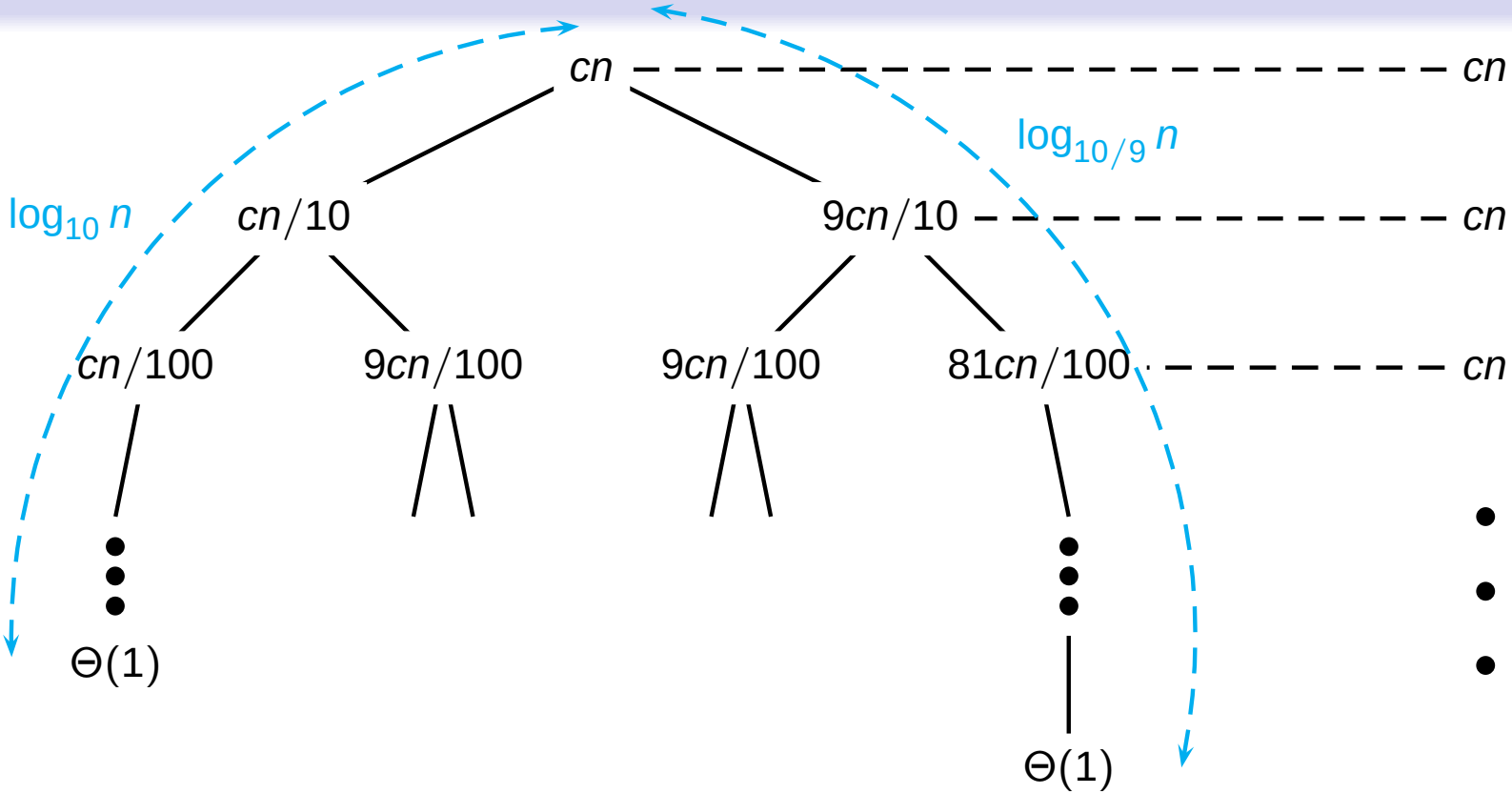
Analysis of “almost-best” case



$$cn \lg_{10} n \leq T(n) \leq cn \lg_{10/9} n$$

$$T(n) \in \Theta(n \lg n)$$

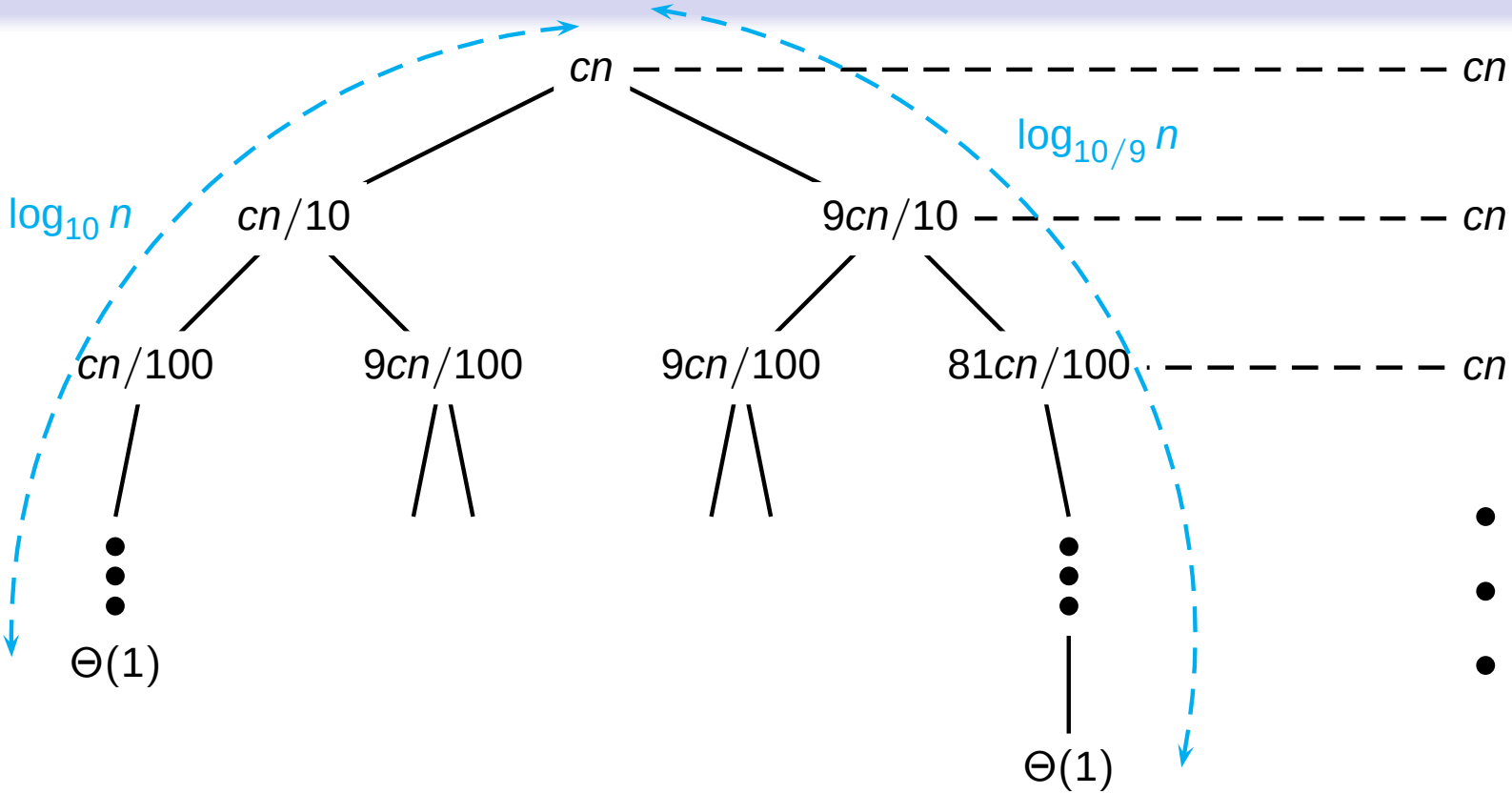
Analysis of “almost-best” case



$$cn \lg_{10} n \leq T(n) \leq cn \lg_{10/9} n$$

$$T(n) \in \Theta(n \lg n)$$

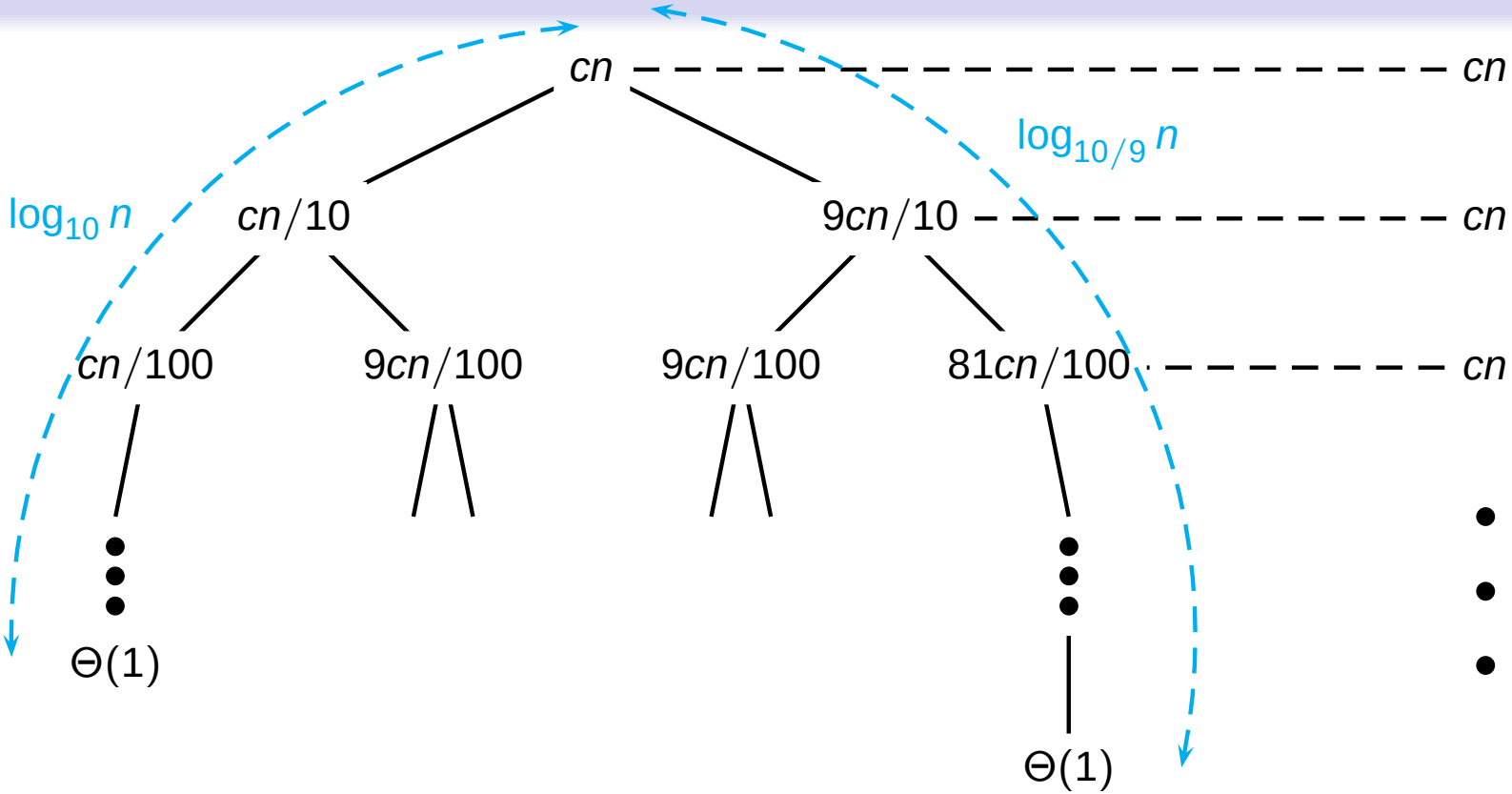
Analysis of “almost-best” case



$$cn \lg_{10} n \leq T(n) \leq cn \lg_{10/9} n$$

$$T(n) \in \Theta(n \lg n)$$

Analysis of “almost-best” case



$$cn \lg_{10} n \leq T(n) \leq cn \lg_{10/9} n$$

$$T(n) \in \Theta(n \lg n)$$

A mixed analysis

Suppose we alternate lucky, unlucky, lucky, unlucky, lucky,...

$$L(n) = 2U(n/2) + \Theta(n) \quad \textbf{lucky}$$
$$U(n) = L(n-1) + \Theta(n) \quad \textbf{unlucky}$$

Solving

$$\begin{aligned} L(n) &= 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) \\ &= 2L(n/2-1) + \Theta(n) \\ &= \Theta(n \lg n) \qquad \text{:-)} \end{aligned}$$

Are we usually lucky?

A mixed analysis

Suppose we alternate lucky, unlucky, lucky, unlucky, lucky,...

$$L(n) = 2U(n/2) + \Theta(n) \quad \textbf{lucky}$$
$$U(n) = L(n-1) + \Theta(n) \quad \textbf{unlucky}$$

Solving

$$\begin{aligned} L(n) &= 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) \\ &= 2L(n/2-1) + \Theta(n) \\ &= \Theta(n \lg n) \qquad \text{:-)} \end{aligned}$$

Are we usually lucky?

A mixed analysis

Suppose we alternate lucky, unlucky, lucky, unlucky, lucky,...

$$L(n) = 2U(n/2) + \Theta(n) \quad \textbf{lucky}$$
$$U(n) = L(n-1) + \Theta(n) \quad \textbf{unlucky}$$

Solving

$$\begin{aligned} L(n) &= 2(L(n/2-1) + \Theta(n/2)) + \Theta(n) \\ &= 2L(n/2-1) + \Theta(n) \\ &= \Theta(n \lg n) \qquad \text{:-)} \end{aligned}$$

Are we usually lucky?

Randomized quicksort

Idea

- Running time is independent of the input order.
- No assumptions need to be made about the input distribution.
- No specific input elicits the worst-case behavior.
- The worst case is determined only by the output of a random-number generator.

Randomized quicksort

RANDOMIZED-PARTITION(A, p, r)

```
1  i = RANDOM(p, r)
2  exchange A[r] ↔ A[i]
3  return PARTITION(A, p, r)
```

RANDOMIZD-QUICKSORT(A, p, r)

```
1  if p < r
2     then q = RANDOMIZED-PARTITON(A, p, r)
3          RANDOMIZD-QUICKSORT(A, p, q − 1)
4          RANDOMIZD-QUICKSORT(A, q + 1, r)
```

Expected time analysis

Let $T(n)$ be the random variable for the running time of RANDOMIZED-QUICKSORT.

For $k = 0, 1, \ldots, n-1$, define the **indicator random variable**

$$X_k = \begin{cases} 1 & \text{if it generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$

$E[X_k] = Pr\{X_k = 1\} = 1/n$, since all splits are equally likely, assuming elements are distinct.

Expected time analysis

Let $T(n)$ be the random variable for the running time of RANDOMIZED-QUICKSORT.

For $k = 0, 1, \ldots, n-1$, define the **indicator random variable**

$$X_k = \begin{cases} 1 & \text{if it generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$

$E[X_k] = Pr\{X_k = 1\} = 1/n$, since all splits are equally likely, assuming elements are distinct.

Expected time analysis

Let $T(n)$ be the random variable for the running time of RANDOMIZED-QUICKSORT.

For $k = 0, 1, \ldots, n-1$, define the **indicator random variable**

$$X_k = \begin{cases} 1 & \text{if it generates a } k : n-k-1 \text{ split,} \\ 0 & \text{otherwise.} \end{cases}$$

$E[X_k] = Pr\{X_k = 1\} = 1/n$, since all splits are equally likely, assuming elements are distinct.

Expected time analysis

$$T(n) = \begin{cases} T(0) + T(n-1) + \Theta(n) & \text{if 0: n-1 split,} \\ T(1) + T(n-2) + \Theta(n) & \text{if 1: n-2 split,} \\ \vdots & \\ T(n-1) + T(0) + \Theta(n) & \text{if n-1: 0 split,} \end{cases}$$

$$= \sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n)).$$

Expected time analysis

$$E[T(n)] = E[\sum_{k=0}^{n-1} X_k(T(k) + T(n-k-1) + \Theta(n))]$$

Take expectations of both sides.

Expected time analysis

$$E[T(n)] = E[\sum_{k=0}^{n-1} X_k(T(k) + T(n-k-1) + \Theta(n))]$$

$$= \sum_{k=0}^{n-1} E[X_k(T(k) + T(n-k-1) + \Theta(n))]$$

Linearity of expectation.

Expected time analysis

$$
\begin{aligned}
E[T(n)] &= E[\sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n))] \\
&= \sum_{k=0}^{n-1} E[X_k (T(k) + T(n-k-1) + \Theta(n))] \\
&= \sum_{k=0}^{n-1} E[X_k] \cdot E[T(k) + T(n-k-1) + \Theta(n)]
\end{aligned}
$$

Independence of X_k from other random choices.

Expected time analysis

$$E[T(n)] = \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)] + \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)$$

Linearity of expections; $E[X_k] = \frac{1}{n}$.

Expected time analysis

$$E[T(n)] = \frac{1}{n}\sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n}\sum_{k=0}^{n-1} E[T(n-k-1)]$$
$$+ \frac{1}{n}\sum_{k=0}^{n-1} \Theta(n)$$
$$= \frac{2}{n}\sum_{k=0}^{n-1} E[T(k)] + \Theta(n)$$

Expected time analysis

$$E[T(n)] = \frac{2}{n}\sum_{k=2}^{n-1} E[T(k)] + \Theta(n)$$

(The $k = 0, 1$ terms can be absorbed in the $\Theta(n)$.)

Prove $E[T(n)] \leq an \lg n$ for constant $a > 0$.

- Choose a large enough so that $an \lg n$ dominates $E[T(n)]$ for sufficiently small $n \geq 2$.

Expected time analysis

Substitution

$$E[T(n)] \leq \frac{2}{n} \sum_{k=2}^{n-1} ak \lg k + \Theta(n)$$

Expected time analysis

Substitution

$$E[T(n)] \leq \frac{2}{n}\sum_{k=2}^{n-1} ak \lg k + \Theta(n)$$
$$\leq \frac{2a}{n}(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2) + \Theta(n)$$

Fact: $\sum_{k=2}^{n-1} k \lg k \leq \frac{1}{2}n^2 \lg n - \frac{1}{8}n^2$.

Expected time analysis

Substitution

$$
\begin{aligned}
E[T(n)] &\leq \frac{2}{n}\sum_{k=2}^{n-1} ak \lg k + \Theta(n) \\
&\leq \frac{2a}{n}\left(\frac{1}{2}n^2 \lg n - \frac{1}{8}n^2\right) + \Theta(n) \\
&= an \lg n - \left(\frac{an}{4} - \Theta(n)\right) \\
&\leq an \lg n
\end{aligned}
$$

Expected time analysis

Lemma 7.1

Let X be the number of comparisons performed in line 4 of PARTITION over the entire execution of QUICKSORT on an n-element array. Then the running time of QUICKSORT is $O(n + X)$.

Expected time analysis

Analysis

- $A = \langle z_1, z_2, \ldots, z_n \rangle$, with z_i being the ith smallest element.
- Define the set $Z = \langle z_i, z_{i+1}, \ldots, z_j \rangle$ to be the set of elements between z_i and z_j.
- $X_{ij} = I\{z_i \text{ is compared to } z_j\}$.
- $X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$

Expected time analysis

Analysis

$$E[X] = E\left[\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right] = \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E[X_{ij}]$$

$$= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} Pr\{z_i \text{ is compared to } z_j\}$$

Expected time analysis

Analysis

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\lg n) = O(n \lg n)$$

Expected time analysis

Analysis

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\lg n) = O(n \lg n)$$

Expected time analysis

Analysis

$$E[X] = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} = \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1}$$

$$< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} = \sum_{i=1}^{n-1} O(\lg n) = O(n \lg n)$$

Quicksort in practice

- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.
- Quicksort can benefit substantially from **code tuning**.
- Quicksort behaves well even with caching and virtual memory.

History

- Sedgewick [1978] and Bentley [1986] provide a good reference on the details of implementation and how they matter.
- M.D. McIlroy [1999] showed how to engineer a “killer adversary” that produces an array on which virtually any implementation of quicksort takes $O(n^2)$.

How fast can we sort?

Definition

All the sorting algorithms we have seen so far are **comparison sorts:** the sorted order is based only on comparisons between the input elements.

The best worst-case running time that we've seen for comparison sorting is $O(n \lg n)$.

Question

Is $O(n \lg n)$ the best we can do?

How fast can we sort?

Definition

All the sorting algorithms we have seen so far are **comparison sorts:** the sorted order is based only on comparisons between the input elements.

The best worst-case running time that we've seen for comparison sorting is $O(n \lg n)$.

Question

Is $O(n \lg n)$ the best we can do?

How fast can we sort?

Definition

All the sorting algorithms we have seen so far are **comparison sorts:** the sorted order is based only on comparisons between the input elements.

The best worst-case running time that we've seen for comparison sorting is $O(n \lg n)$.

Question

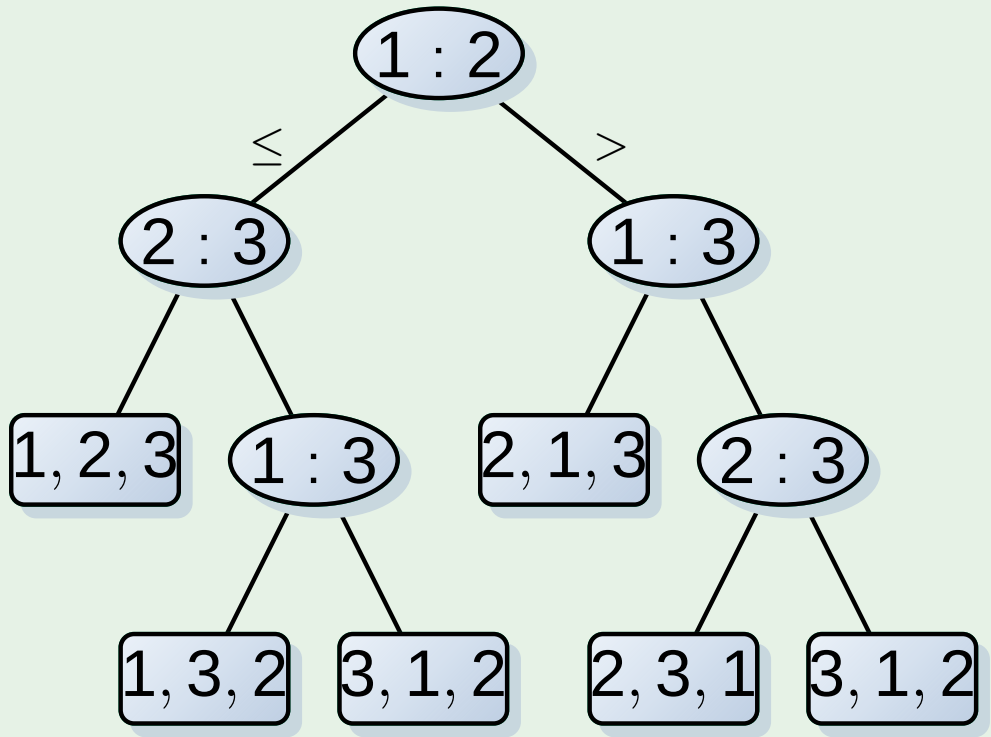
Is $O(n \lg n)$ the best we can do?

Decision-tree example

Each internal node is labeled $i : j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i > a_j$.

Example



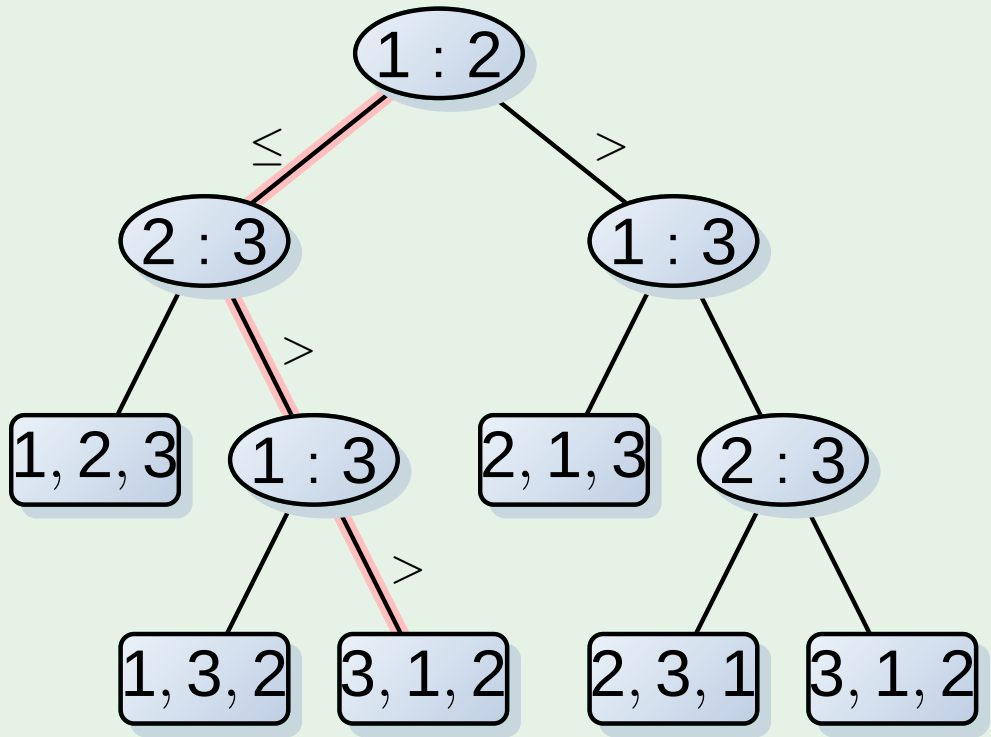
Sort $\langle a_1, a_2, a_3 \rangle = \langle 6, 8, 5 \rangle$

Decision-tree example

Each internal node is labeled $i : j$ for $i, j \in \{1, 2, \ldots, n\}$.

- The left subtree shows subsequent comparisons if $a_i \leq a_j$.
- The right subtree shows subsequent comparisons if $a_i > a_j$.

Example



Sort $\langle a_1, a_2, a_3 \rangle = \langle 6, 8, 5 \rangle$

Decision-tree model

A decision tree can model the execution of any comparison sort:

- One tree for each input size n.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.

Decision-tree model

A decision tree can model the execution of any comparison sort:

- One tree for each input size n.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.

Decision-tree model

A decision tree can model the execution of any comparison sort:

- One tree for each input size n.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.

Decision-tree model

A decision tree can model the execution of any comparison sort:

- One tree for each input size n.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.

Decision-tree model

A decision tree can model the execution of any comparison sort:

- One tree for each input size n.
- View the algorithm as splitting whenever it compares two elements.
- The tree contains the comparisons along all possible instruction traces.
- The running time of the algorithm = the length of the path taken.
- Worst-case running time = height of tree.

Lower bound for decision-tree sorting

Theorem

Any decision tree that can sort n elements must have height $\Omega(n \lg n)$.

Lower bound for decision-tree sorting

Proof.

The tree must contain $\geq n!$ leaves, since there are $n!$ possible permutations. A height-h binary tree has $\leq 2^h$ leaves. Thus, $n! \leq 2^h$.

$$\begin{aligned} \therefore h &\geq \lg(n!) \\ &\geq \lg((n/e)^n) \qquad \text{(Stirling's formula)} \\ &= n \lg n - n \lg e \\ &= \Omega(n \lg n). \end{aligned}$$

Counting sort

Definition

- **Assumption:** Each of n input elements is an integer in the range between 1 and k.
- **Aim:** Runs in $\Theta(n)$ time.
- **Input:** $A[1 \ldots n]$, where $A[j] \in \{1, 2, \ldots, k\}$.
- **Output:** $B[1 \ldots n]$, sorted.
- **Auxiliary storage:** $C[1 \ldots k]$.

Counting sort

COUNTING-SORT(A, B, K)

```
1  for i = 0 to k
2        do C[i] = 0
3  for j = 1 to A.length
4        do C[A[j]] = C[A[j]] + 1
   ▷ C[i] now contains the number of elements equal to i.
5  for i = 1 to k
6        do C[i] = C[i] + C[i − 1]
   ▷ C[i] now contains the number of elements
   ▷ less than or equal to i.
7  for j = A.length downto 1
8        do B[C[A[j]]] = A[j]
9           C[A[j]] = C[A[j]] − 1
```

Counting sort

Example

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

for $j = 1$ **to** $A.length$
do $C[A[j]] = C[A[j]] + 1$

Counting sort

Example

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

for $j = 1$ **to** $A.length$
do $C[A[j]] = C[A[j]] + 1$

Counting sort

Example

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

	0	1	2	3	4	5
C'	2	2	4	7	7	8

for $i = 1$ **to** k
do $C[i] = C[i] + C[i-1]$

Counting sort

Example

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

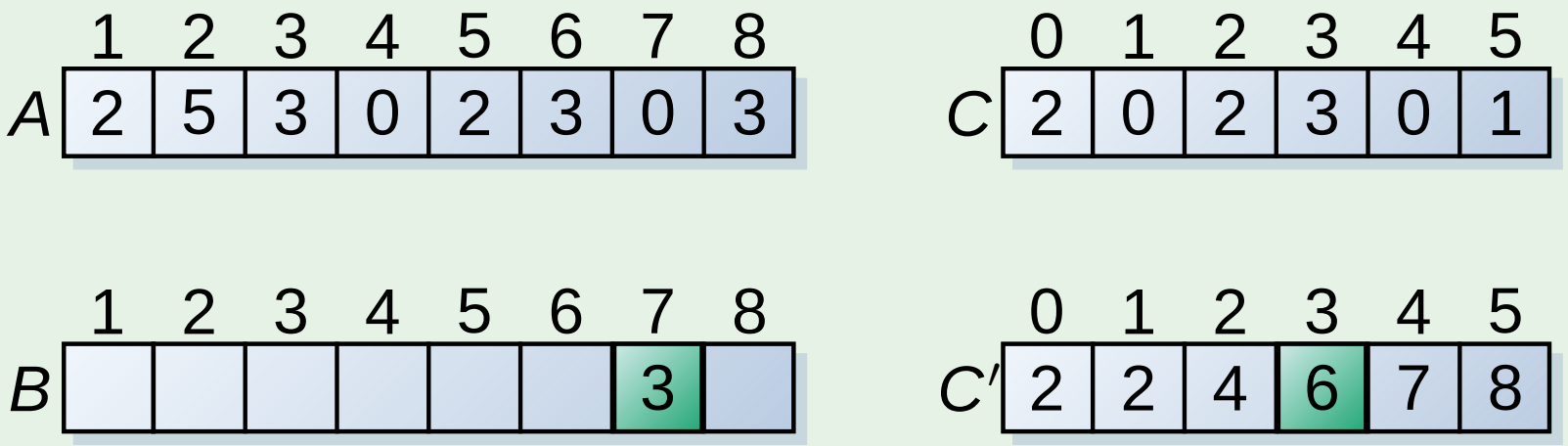
	0	1	2	3	4	5
C	2	0	2	3	0	1

	0	1	2	3	4	5
C'	2	2	4	7	7	8

for $j = A.length$ **downto** 1
do $B[C[A[j]]] = A[j]$
$C[A[j]] = C[A[j]] - 1$

Counting sort

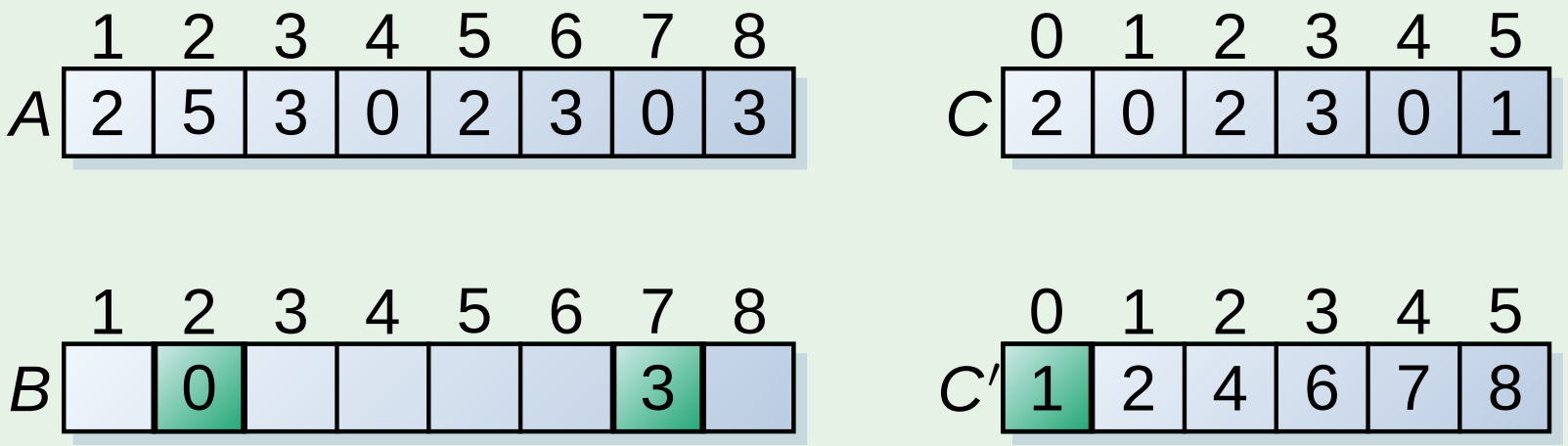
Example



for $j = A.length$ **downto** 1
do $B[C[A[j]]] = A[j]$
$C[A[j]] = C[A[j]] - 1$

Counting sort

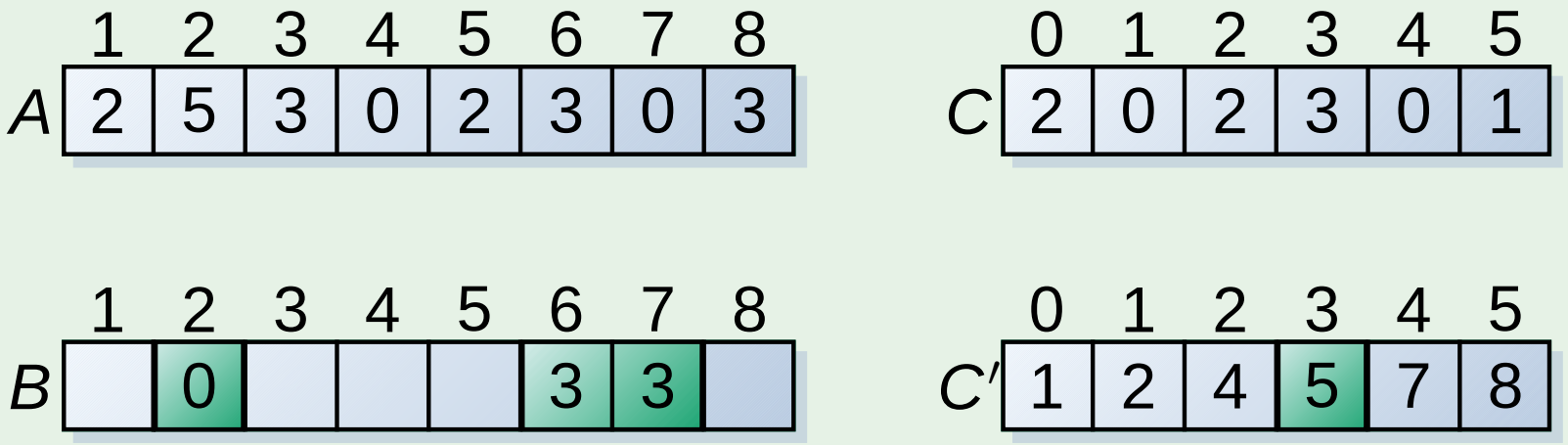
Example



for $j = A.length$ **downto** 1
do $B[C[A[j]]] = A[j]$
$C[A[j]] = C[A[j]] - 1$

Counting sort

Example



for $j = A.length$ **downto** 1
do $B[C[A[j]]] = A[j]$
$C[A[j]] = C[A[j]] - 1$

Counting sort

Example

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

	1	2	3	4	5	6	7	8
B	0	0	2	2	3	3	3	5

	0	1	2	3	4	5
C'	0	2	2	4	7	7

for $j = A.length$ **downto** 1
do $B[C[A[j]]] = A[j]$
$C[A[j]] = C[A[j]] - 1$

Analysis

Running time

If $k = O(n)$, then counting sort takes $\Theta(n)$ time.

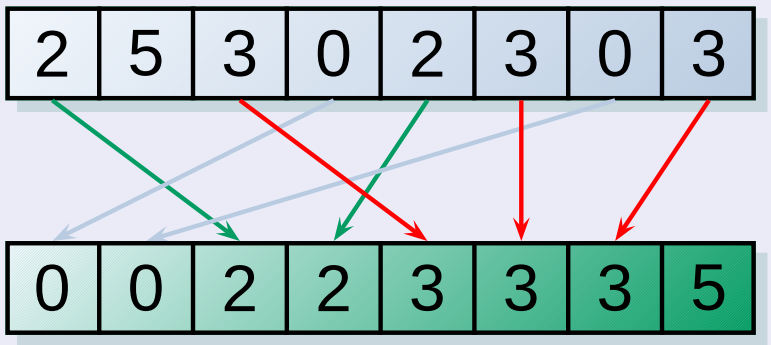
Why?

- **Comparison sorting** takes $\Omega(n \lg n)$ time.
- Counting sort is **not** a **comparison sort**.

Analysis

Stable sorting

Counting sort is a stable sort: it preserves the input order among equal elements.



Radix sort

Idea

- Digit-by-Digit sort.
- Bad idea: Sort on most-significant digit first.
- Good idea: Sort on **least-significant digit first** with auxiliary **stable** sort.

Radix sort

Idea

- Digit-by-Digit sort.
- Bad idea: Sort on most-significant digit first.
- Good idea: Sort on **least-significant digit first** with auxiliary **stable** sort.

Radix sort

Idea

- Digit-by-Digit sort.
- Bad idea: Sort on most-significant digit first.
- Good idea: Sort on **least-significant digit first** with auxiliary **stable** sort.

Radix sort

Example

3 2 9

4 5 7

6 5 7

8 3 9

4 3 6

7 2 0

3 5 5

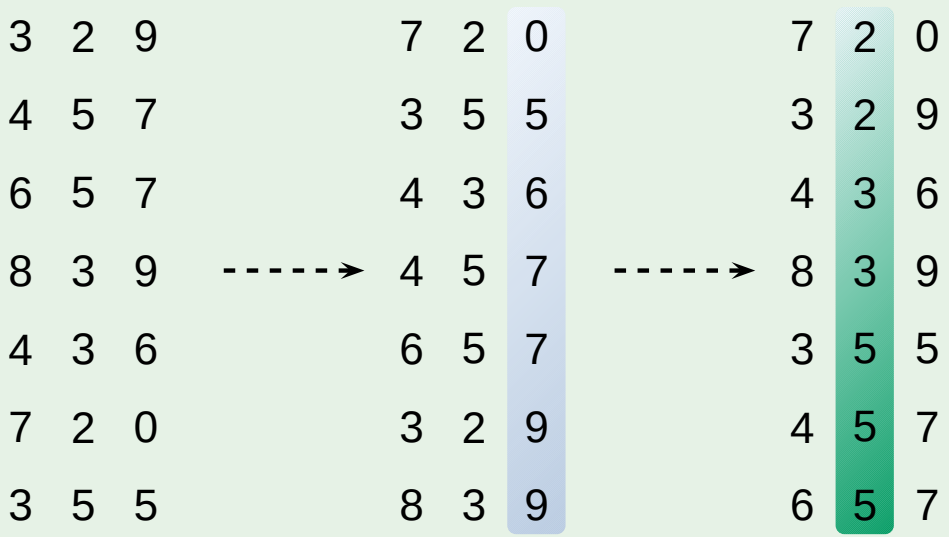
Radix sort

Example

3	2	9		7	2	0
4	5	7		3	5	5
6	5	7		4	3	6
8	3	9	------→	4	5	7
4	3	6		6	5	7
7	2	0		3	2	9
3	5	5		8	3	9

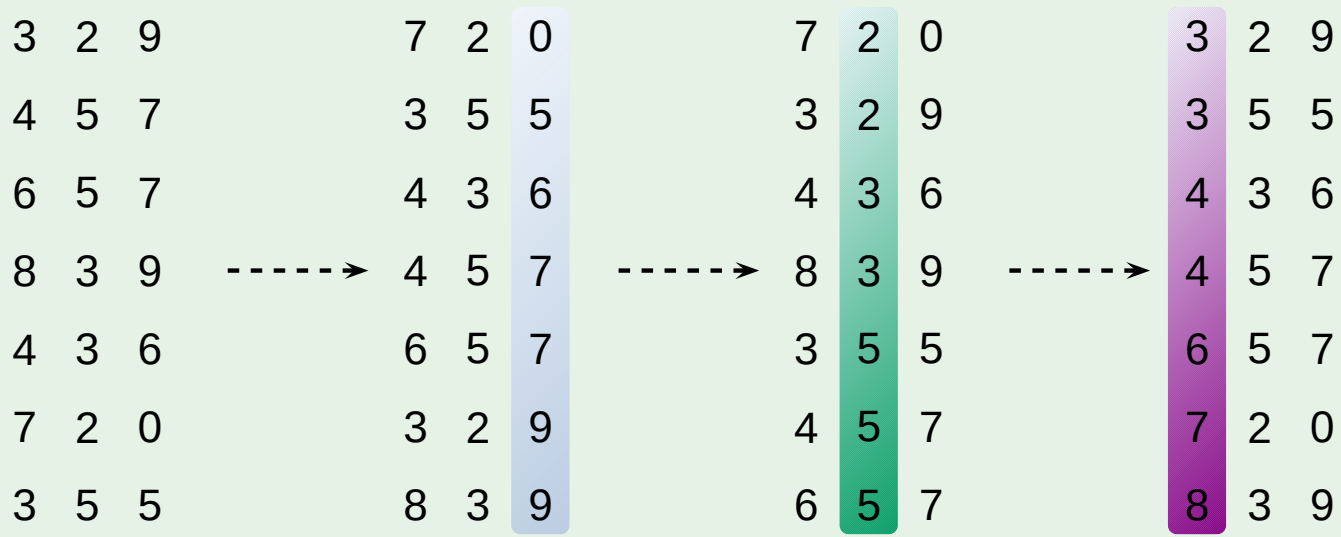
Radix sort

Example



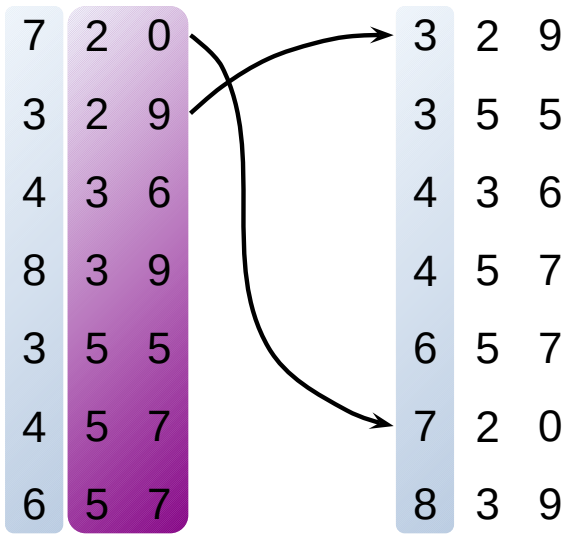
Radix sort

Example



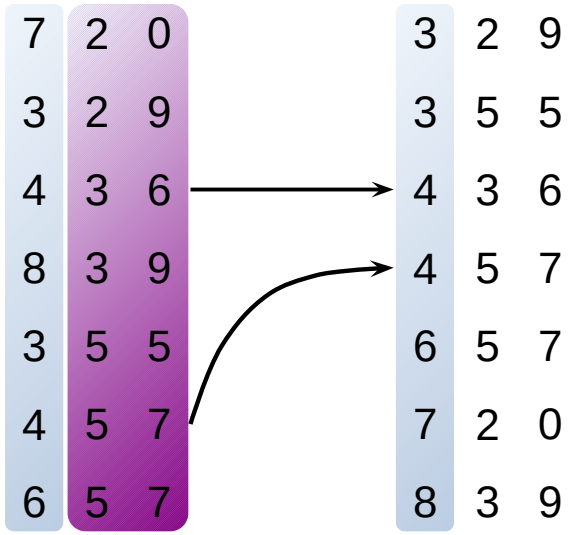
Correctness of radix sort

- Assume that the numbers are sorted by their low-order $t-1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.
  - Two numbers equal in digit t are put in the same order as the input



Correctness of radix sort

- Assume that the numbers are sorted by their low-order $t-1$ digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.
  - Two numbers equal in digit t are put in the same order as the input



Analysis of radix sort

Analysis

- Assume counting sort is the auxiliary stable sort.
- Sort n computer words of b bits each.
- Each word can be viewed as having b/r base-2^r digits.

Analysis of radix sort

Example

Sorting a 32-bit word.

- $r = 8 \Rightarrow b/r = 4$ passes of counting sort on base-2^8 digits;
- $r = 16 \Rightarrow b/r = 2$ passes of counting sort on base-2^{16} digits.

How many passes should we make?

Analysis of radix sort

Example

Sorting a 32-bit word.

- $r = 8 \Rightarrow b/r = 4$ passes of counting sort on base-2^8 digits;
- $r = 16 \Rightarrow b/r = 2$ passes of counting sort on base-2^{16} digits.

How many passes should we make?

Analysis of radix sort

Analysis

- Counting sort takes $\Theta(n+k)$ time to sort n numbers in the range from 0 to $k-1$.
- If each b-bit word is broken into r-bit pieces, each pass of counting sort takes $\Theta(n+2^r)$ time.

Analysis of radix sort

Analysis

- Counting sort takes $\Theta(n+k)$ time to sort n numbers in the range from 0 to $k-1$.
- If each b-bit word is broken into r-bit pieces, each pass of counting sort takes $\Theta(n+2^r)$ time.

Analysis of radix sort

Analysis

- Since there are b/r passes, we have

$$T(n,b) = \Theta(\frac{b}{r}(n+2^r))$$

Choose r to minimize $T(n,b)$.

- Choosing $r = \lg n$ implies $T(n,b) = \Theta(bn/\lg n)$.

Analysis of radix sort

Analysis

- Since there are b/r passes, we have

$$T(n,b) = \Theta(\frac{b}{r}(n+2^r))$$

Choose r to minimize $T(n,b)$.

- Choosing $r = \lg n$ implies $T(n,b) = \Theta(bn/\lg n)$.

Analysis of radix sort

Analysis

- In practice, radix sort is fast for large inputs, as well as simple to code and maintain.
- Sorting 32-bit numbers:
  - At most 3 passes when sorting ≥ 2000 numbers.
  - Merge sort and quicksort do at least $\lceil \lg 2000 \rceil = 11$ passes.

Analysis of radix sort

Analysis

- Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies.

Bucket Sort

Idea

Assume the input array A is generated by a random process that distributes elements uniformly over the interval $[0, 1)$.

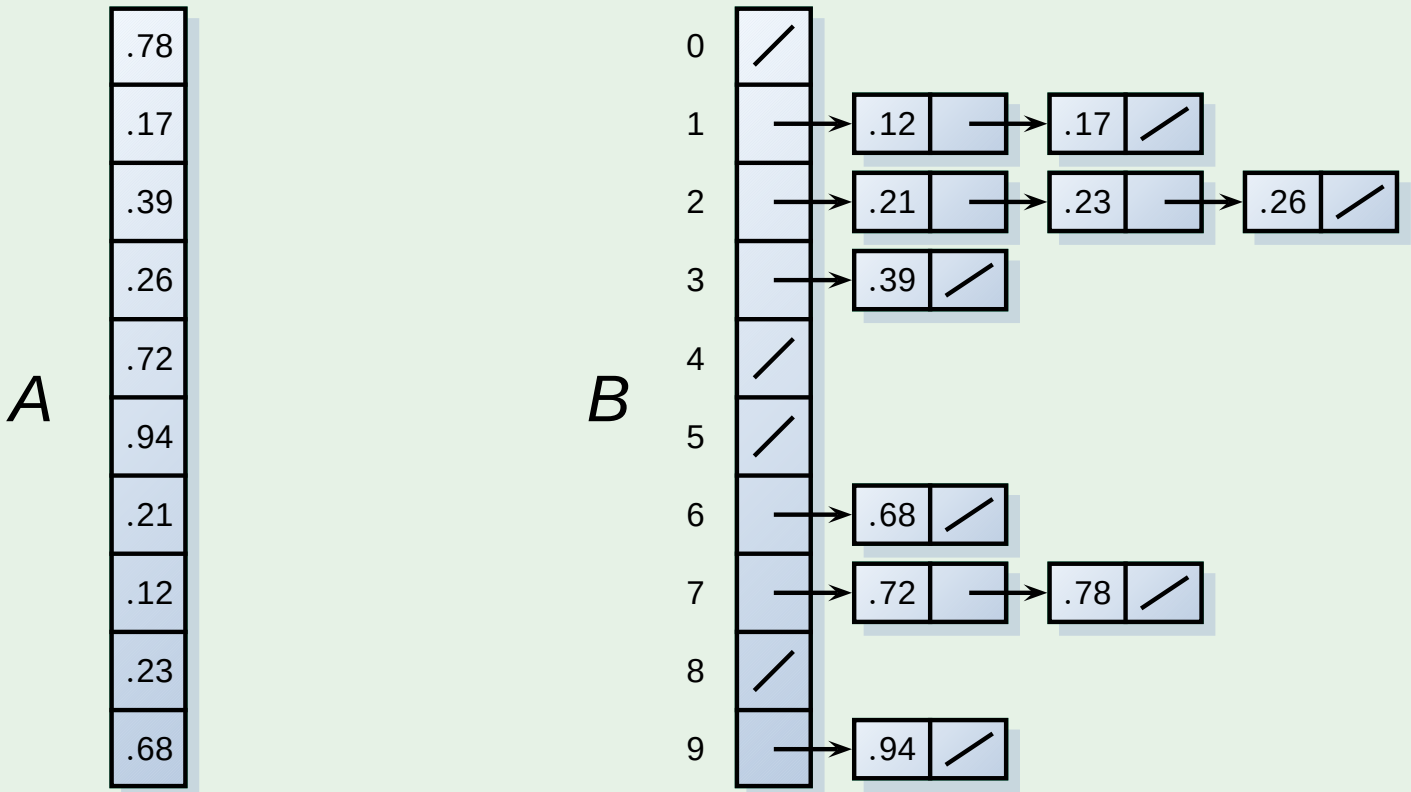
Bucket Sort

BUCKET-SORT(A)

1 $n = A.length$
2 **for** $i = 1$ **to** n
3 **do** insert $A[i]$ into list $B[\lfloor nA[i] \rfloor]$
4 **for** $i = 0$ **to** $n - 1$
5 **do** sort list B[i] with insertion sort
6 concatenate the lists $B[0], B[1], ..., B[n-1]$ together in order

Bucket Sort

Example



Analysis of bucket sort

Let n_i be the random variable denoting the number of elements placed in bucket $B[i]$. Since insertion sort runs in quadratic time, the running time of bucket sort is

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)$$

Analysis of bucket sort

Taking expectations of both sides and using linearity of expectation, we have

$$\begin{aligned} E[T(n)] &= E[\Theta(n) + \sum_{i=0}^{n-1} O(n_i^2)] \\ &= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \end{aligned}$$

Analysis of bucket sort

We define indicator random variables, $X_{ij} = I\{A[j] \text{ falls in bucket } i\}$ for $i = 0, 1, \ldots, n-1$ and $j = 1, 2, \ldots, n$. Thus, $n_i = \sum_{j=1}^{n} X_{ij}$.

$$E[n_i^2] = E[(\sum_{j=1}^{n} X_{ij})^2] = E[\sum_{j=1}^{n}\sum_{k=1}^{n} X_{ij}X_{ik}]$$

$$= E[\sum_{j=1}^{n} X_{ij}^2 + \sum_{1\le j\le n}\sum_{1\le k\le n, k\neq j} X_{ij}X_{ik}]$$

Analysis of bucket sort

Then,

$$E[n_i^2] = \sum_{j=1}^{n} E[X_{ij}^2] + \sum_{1 \le j \le n} \sum_{1 \le k \le n, k \ne j} E[X_{ij} X_{ik}],$$

where $E[X_{ij}^2] = 1/n$ and $E[X_{ij} X_{ik}] = 1/n^2$. We obtain

$$\begin{aligned} E[n_i^2] &= \sum_{j=1}^{n} \frac{1}{n} + \sum_{1 \le j \le n} \sum_{1 \le k \le n, k \ne j} \frac{1}{n^2} \\ &= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} = 2 - \frac{1}{n} \end{aligned}$$

Analysis of bucket sort

$$
\begin{aligned}
E[T(n)] &= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \\
&= \Theta(n) + n \cdot O(2 - 1/n) \\
&= \Theta(n)
\end{aligned}
$$

As long as the input has the property that the sum of the squares of the bucket sizes is linear in the total number of elements, the bucket sort will run in **linear time**.

State-of-the-art of sorting

- In 1993, Fredman and Willard introduced the **fusion tree** data structure and used it to sort n integers in $O(n \lg n / \lg \lg n)$. This bound was later improved to $O(n\sqrt{\lg n})$ time by Anddersson in 1996.
- In 1998, Thorup gave an $O(n(\lg \lg n)^2)$-time sorting algorithm that does not use multiplication or randomization, and uses linear space.

State-of-the-art of sorting

- In 2001, Han improved the bound for sorting to $n \lg \lg n \lg \lg \lg n$ time by combining these techniques.
- However, all of these algorithms are fairly complicated and seem unlikely to compete with existing sorting algorithms in practice.