COMP 446 / 546 ALGORITHM DESIGN AND ANALYSIS

LECTURE 6 LINEAR-TIME SORTING

ALPTEKİN KÜPÇÜ

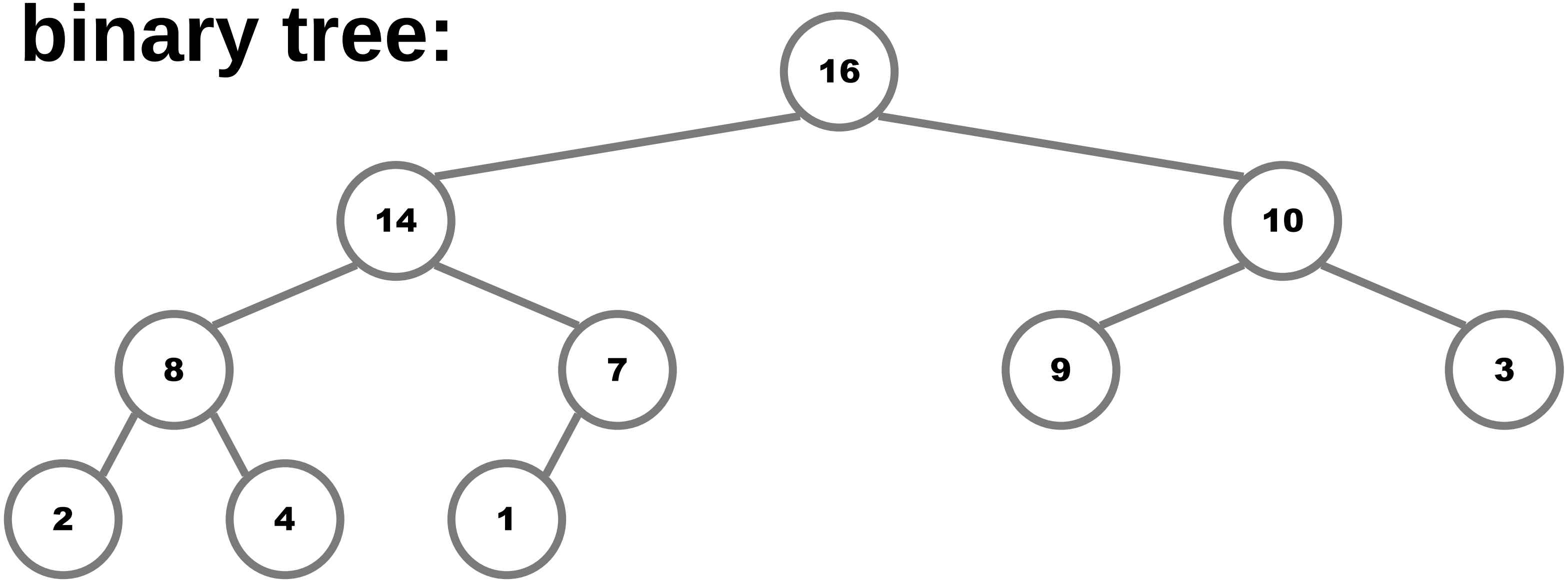
Based on slides of David Luebke, Jennifer Welch, Michael Goodrich, Roberto Tamassia, and Cevdet Aykanat

SORTING REVISITED

- **So far:**
  - **Quicksort**
    - $O(n^2)$ worst-case, $O(n \log n)$ average-case
    - $O(n \log n)$ expected time for Randomized Quicksort
  - **Merge Sort**
    - $O(n \log n)$ worst-case running time
  - **Insertion Sort**
    - Sorts in-place
    - $O(n^2)$ worst-case but $O(n)$ best-case
- **Next: Heapsort**
  - Combines advantages of Merge Sort and Insertion Sort
    - $O(n \log n)$ worst-case, in-place
  - Another design paradigm

HEAP

- A ***heap*** **can be seen as a nearly-complete binary tree:**

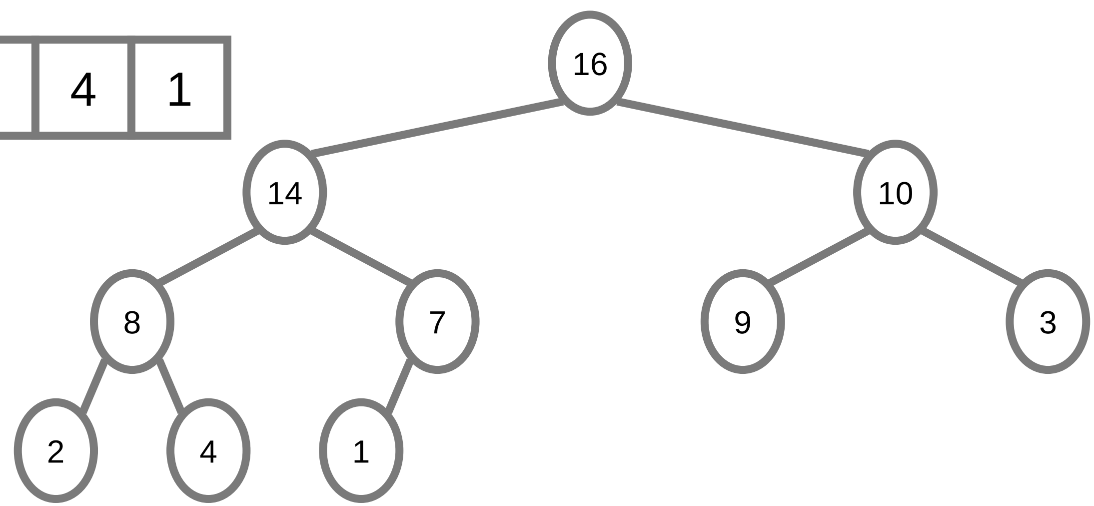


- ***What makes a binary tree complete?***
- ***Is the example above complete?***

ARRAY REPRESENTATION

- **Represent a nearly-complete binary tree as an array:**
  - The root node is the first element A[1]
  - Node *i* is A[*i*]
  - The parent of node *i* is A[*i*/2] (integer division)
  - The left child of node *i* is A[2*i*]
  - The right child of node *i* is A[2*i* + 1]

16	14	10	8	7	9	3	2	4	1

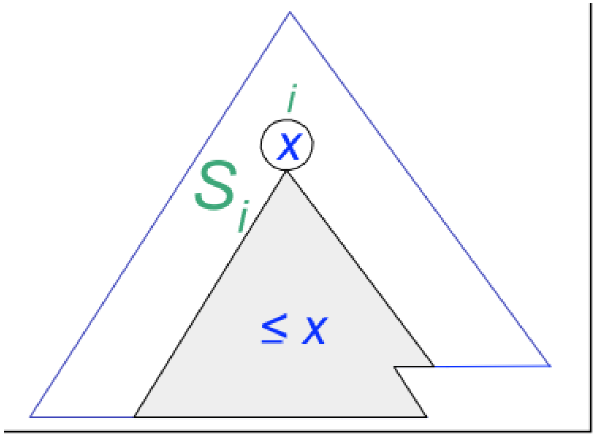


HEAP PROPERTY

- **Heaps also satisfy the *heap property*:**

  A[*Parent*(*i*)] ≥ A[*i*]   for all nodes *i* > 1

  - The value of a node is at most the value of its parent

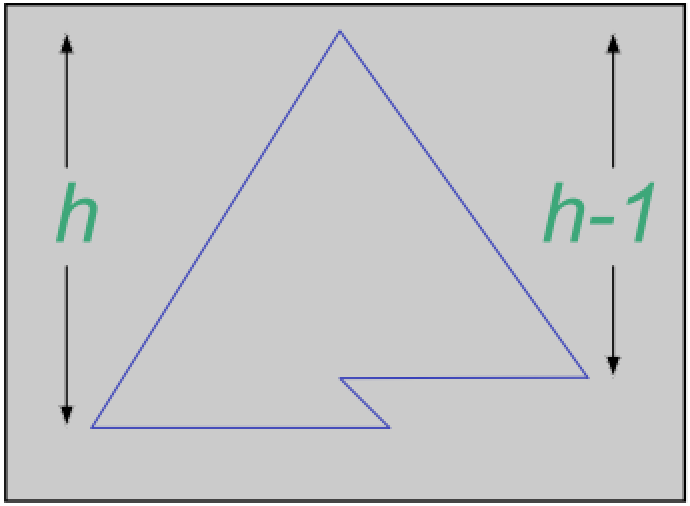


max-heap

- ***Where is the largest element in a heap stored?***
  - Largest element in a sub-tree of a heap is at the root of the sub-tree.
- **For a min-heap, the relation would be otherwise:**
  - A[*Parent*(*i*)] ≤ A[*i*]

HEAP HEIGHT

- The *height* of a node in the tree is the number of edges on the longest (leftmost) path to a leaf
- The height of a tree is the height of its root
- ***What is the height of an n-element heap?***



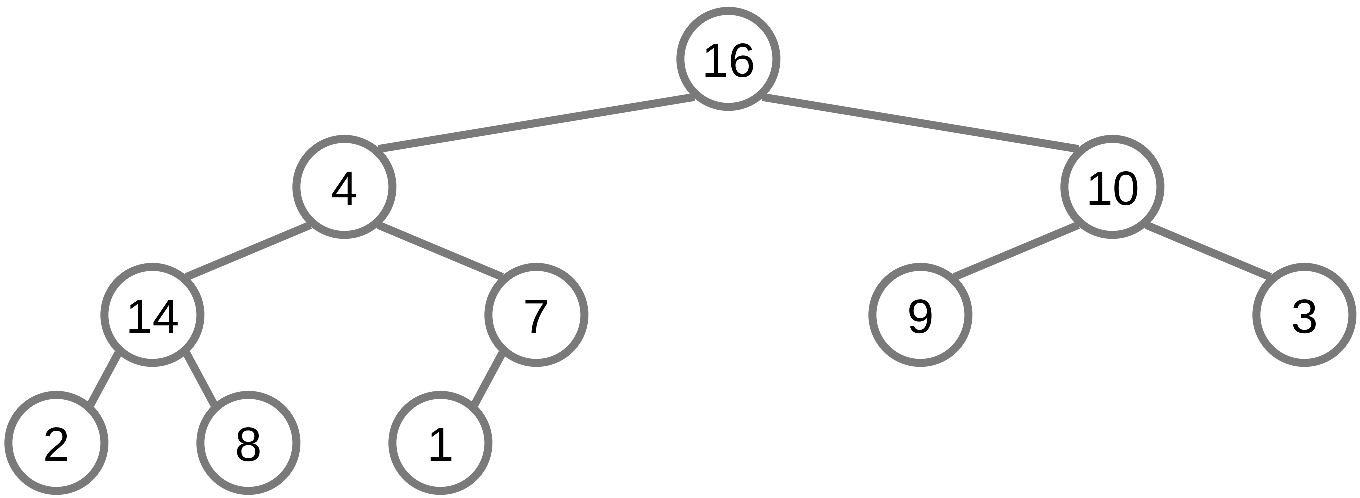
HEAP OPERATIONS: HEAPIFY()

- `HEAPIFY(i)`: **maintain the heap property**
  - **Given:** a node *i* in the heap with left child *l* and right child *r*
  - **Given:** two sub-trees rooted at *l* and *r*, assumed to be heaps
  - **Problem:** The sub-tree rooted at *i* may violate the heap property ***(How?)***
  - **Action:** let the value of the parent node "float down" so sub-tree rooted at *i* satisfies the heap property
    - ***What will be the basic operation between i, l, and r?***

HEAP OPERATIONS: HEAPIFY()

```
HEAPIFY(A, i)
    l = Left(i)
    r = Right(i)
    largest = indexof(max(A[i], A[l], A[r]))
    if (largest != i) then
        swap A[i] ↔ A[largest]
        HEAPIFY(A, largest)
```

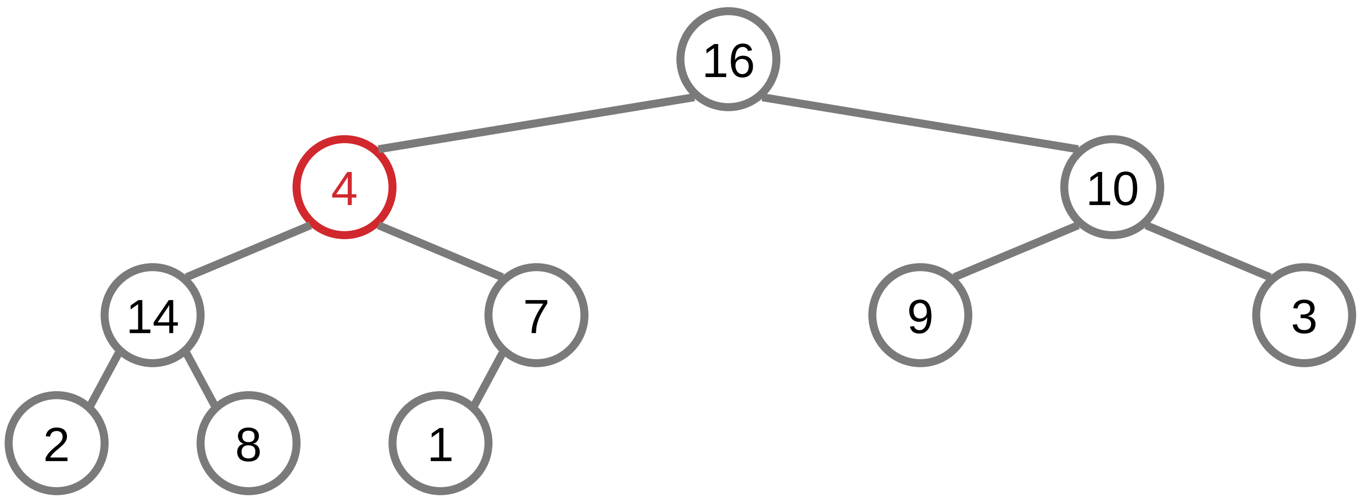
HEAPIFY() EXAMPLE



A =

16	4	10	14	7	9	3	2	8	1

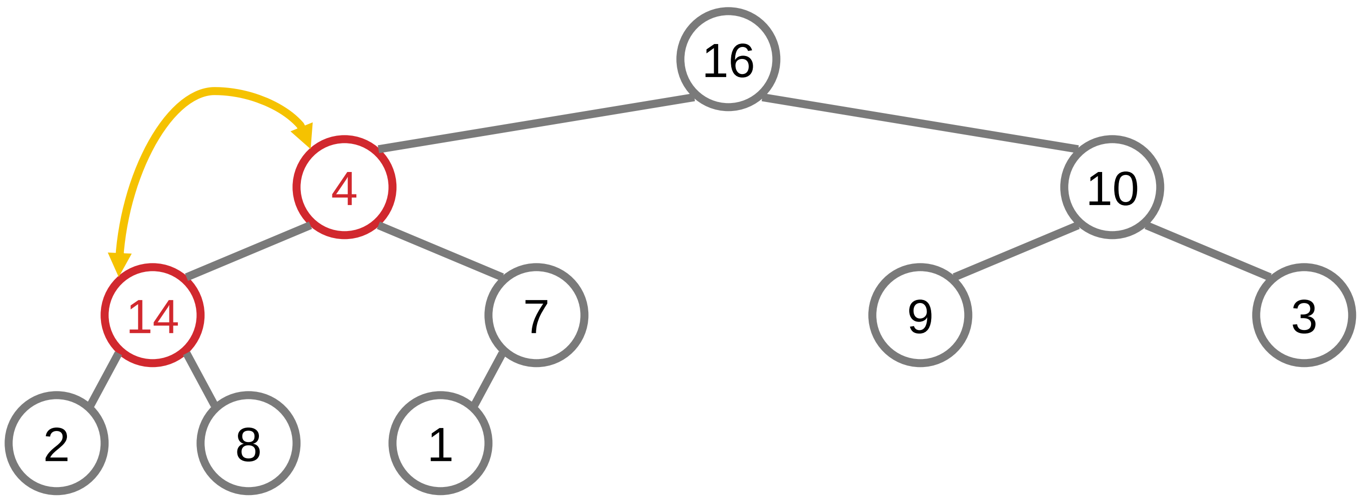
HEAPIFY() EXAMPLE



A =

16	4	10	14	7	9	3	2	8	1

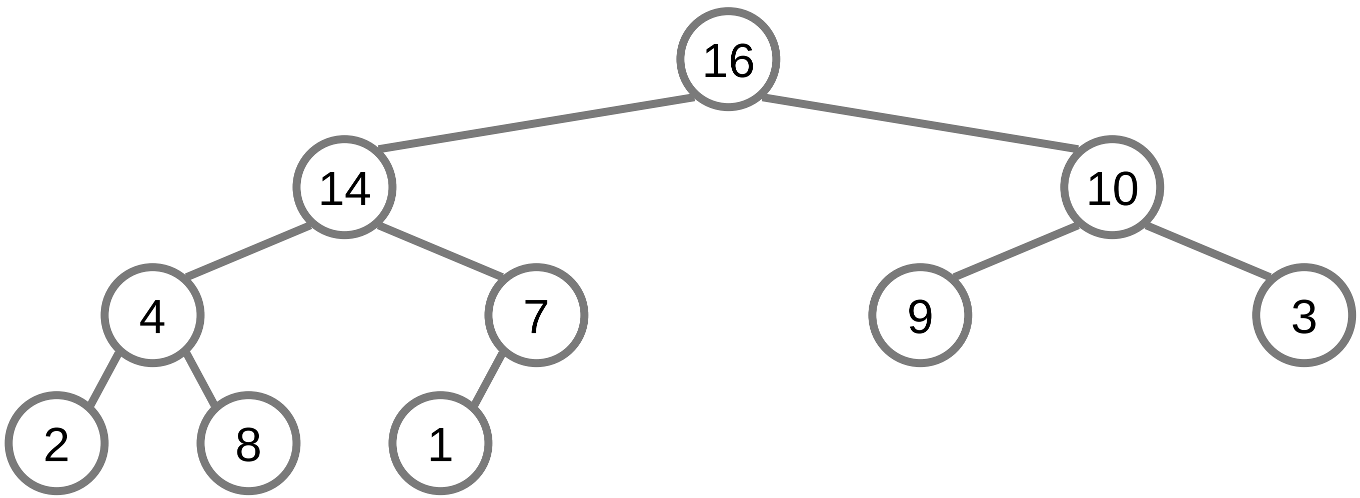
HEAPIFY() EXAMPLE



A =

16	4	10	14	7	9	3	2	8	1

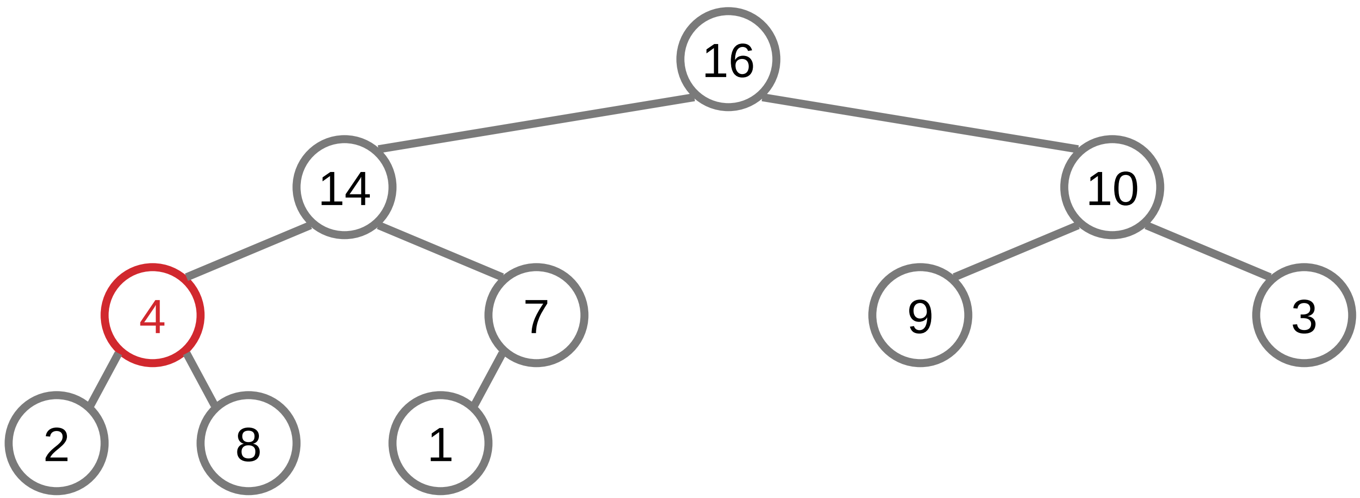
HEAPIFY() EXAMPLE



A =

16	14	10	4	7	9	3	2	8	1

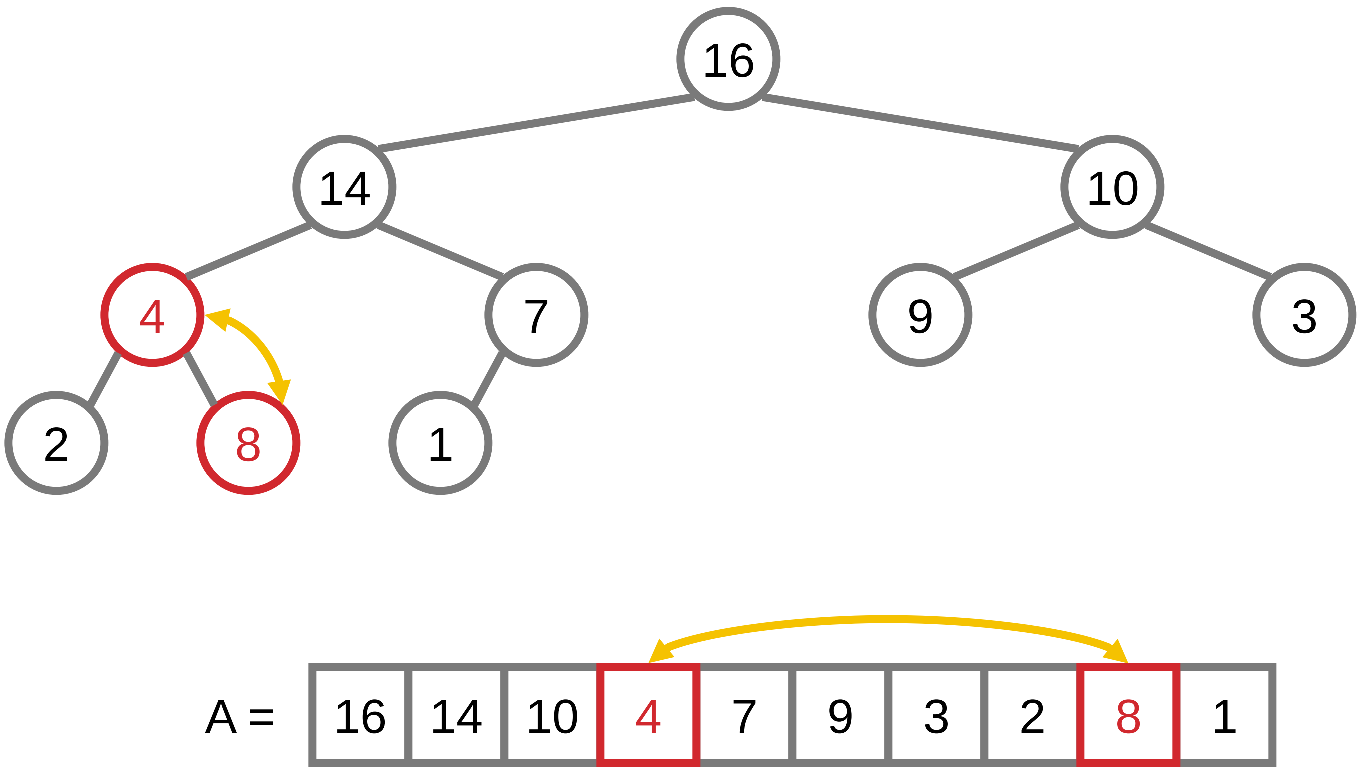
HEAPIFY() EXAMPLE



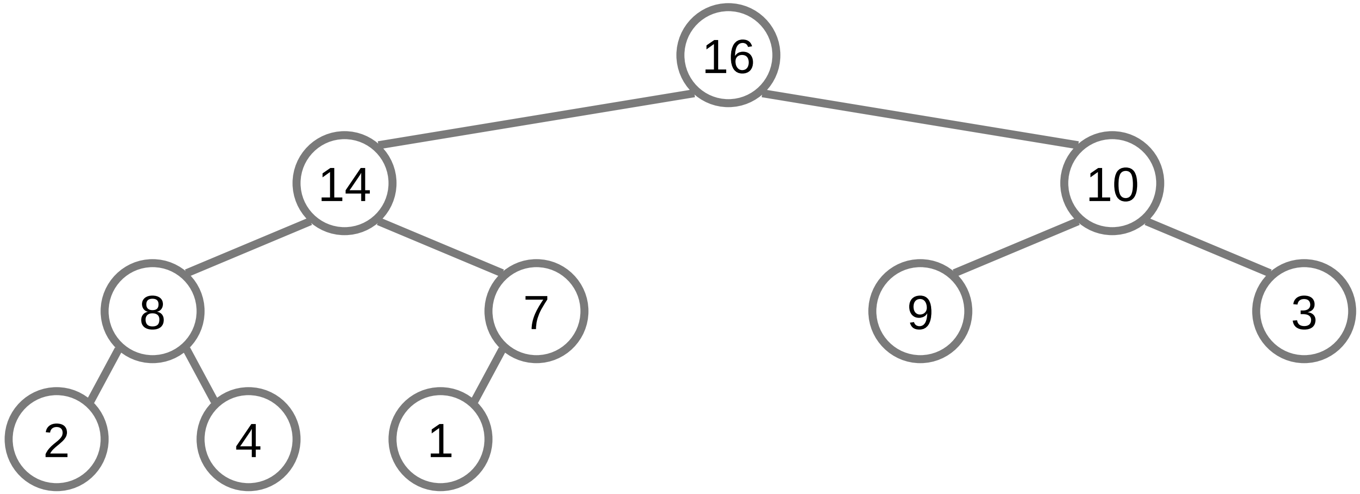
A =

16	14	10	4	7	9	3	2	8	1

HEAPIFY() EXAMPLE

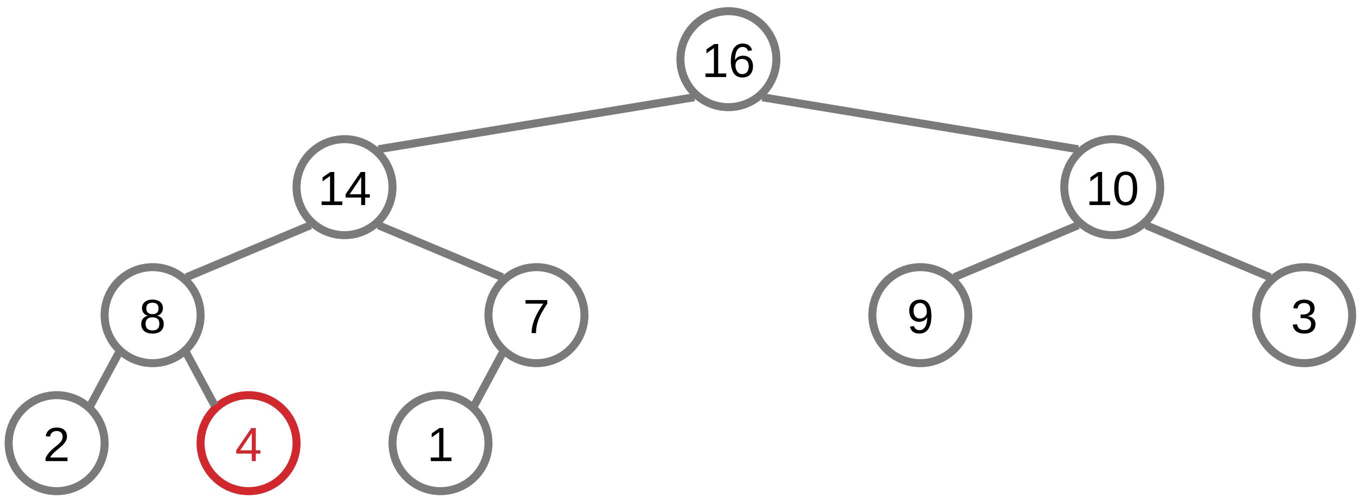


HEAPIFY() EXAMPLE



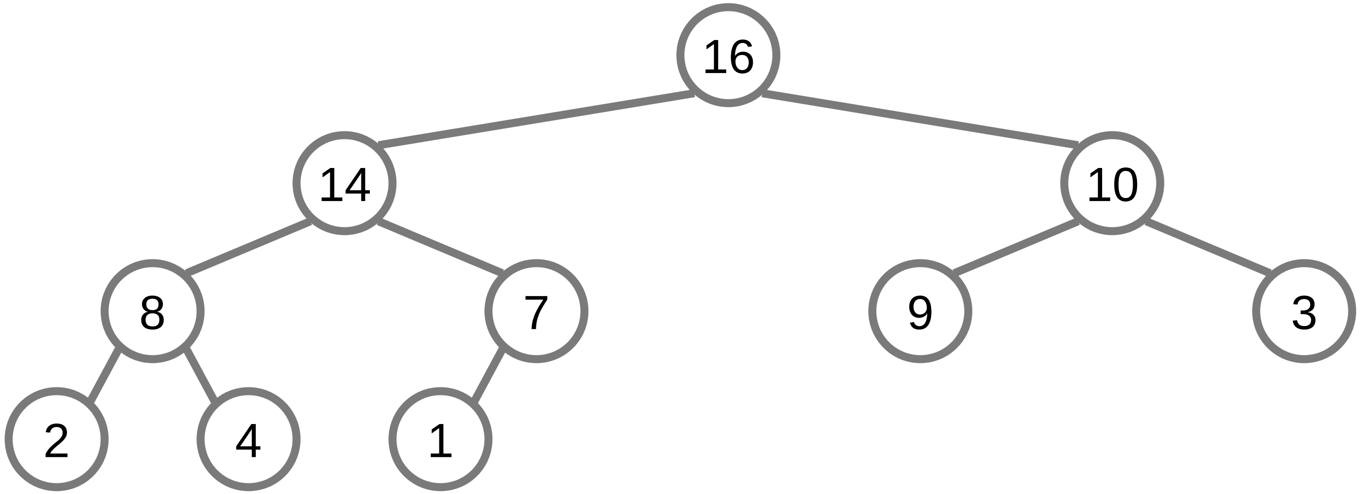
A =	16	14	10	8	7	9	3	2	4	1

HEAPIFY() EXAMPLE



A =	16	14	10	8	7	9	3	2	4	1

HEAPIFY() EXAMPLE



A =

16	14	10	8	7	9	3	2	4	1

HEAPIFY() RUNNING TIME

- **Within a single recursive call, what is the running time of `HEAPIFY()`?**

- **How many times can `HEAPIFY()` recursively call itself in the worst-case?**

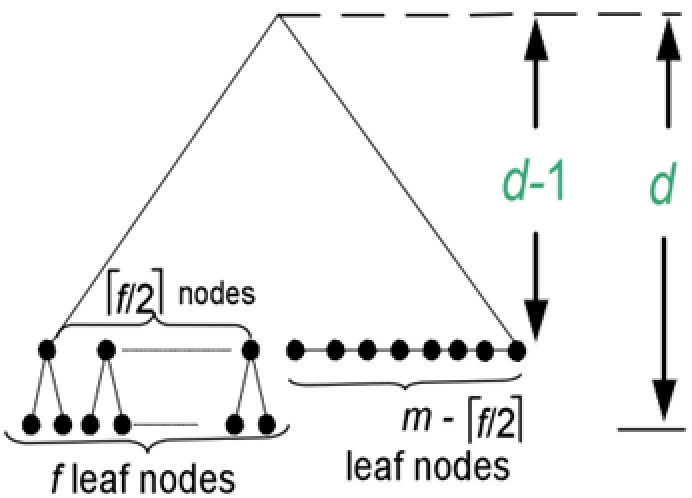
- **What is the worst-case running time of `HEAPIFY()` on a heap of size n?**

HEAPIFY() RUNNING TIME

- **Within a single recursive call, what is the running time of `HEAPIFY()`?**
  - O(1)
- **How many times can `HEAPIFY()` recursively call itself in the worst-case?**
  - O(height) = O(log n)
- **What is the worst-case running time of `HEAPIFY()` on a heap of size n?**
  - O(log n)

HEAP OPERATIONS: BUILDHEAP()

- **Build a heap in a bottom-up manner by running `HEAPIFY()` on successive sub-trees**
- **For array of length *n*, all elements in range A[$\lfloor n/2 \rfloor$ + 1 ... n] are already heaps (*Why?*)**



All leaves are heaps by default
Denote #nodes at level d-1 by m
$m = 2^{d-1}$
Total #nodes is n
$n = 2^{d+1} - 1 - 2(m-f/2)$
$= 4m - 1 - 2m + f$
$= 2m + f - 1$
#leafs = $m - f/2 + f = \lceil n/2 \rceil$

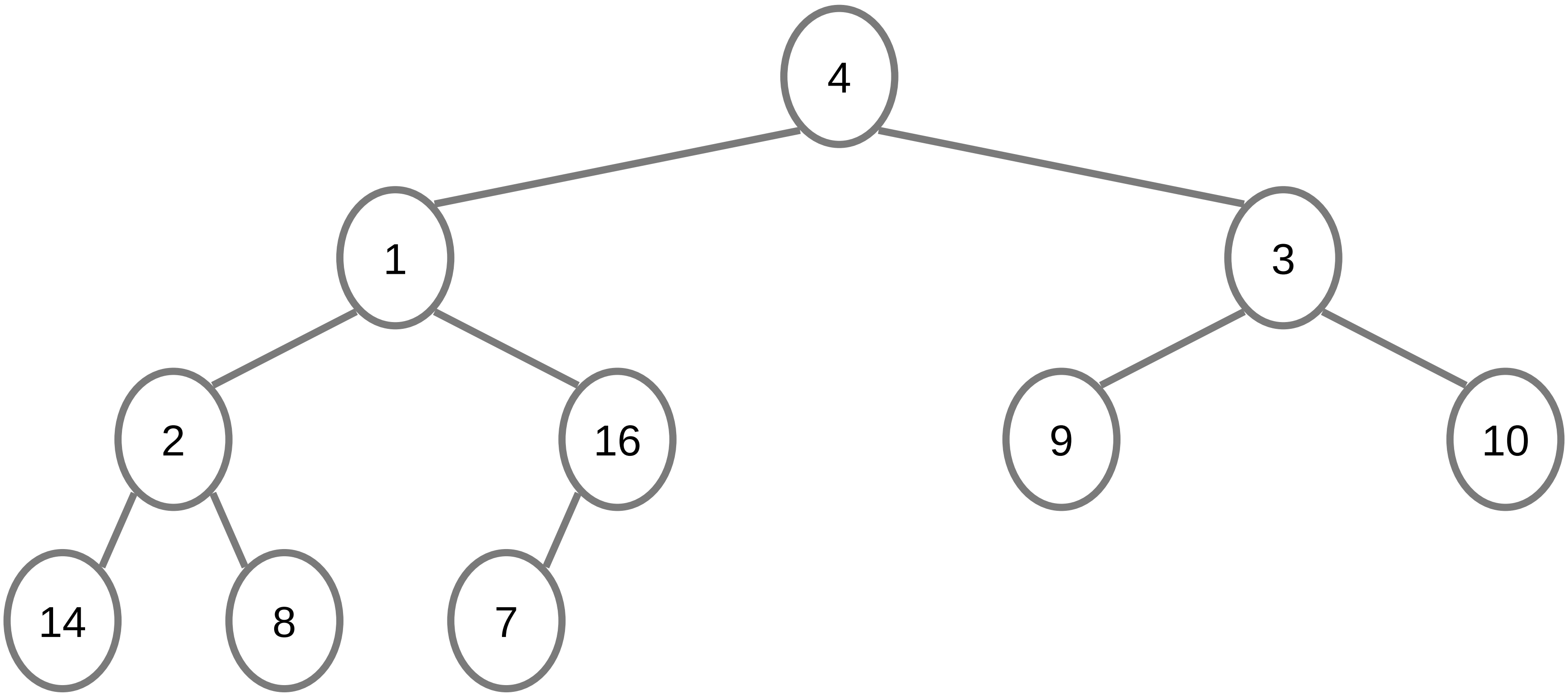
HEAP OPERATIONS: BUILDHEAP()

- **Walk backwards through the array from n/2 to 1, calling `HEAPIFY()` on each node.**
- **Order of processing guarantees that the children of node *i* are already heaps when *i* is processed during `HEAPIFY(i)`**

```
BUILDHEAP (A, n)
      for (i = ⌊n/2⌋ downto 1)
          HEAPIFY(A, i)
```

BUILDHEAP() EXAMPLE

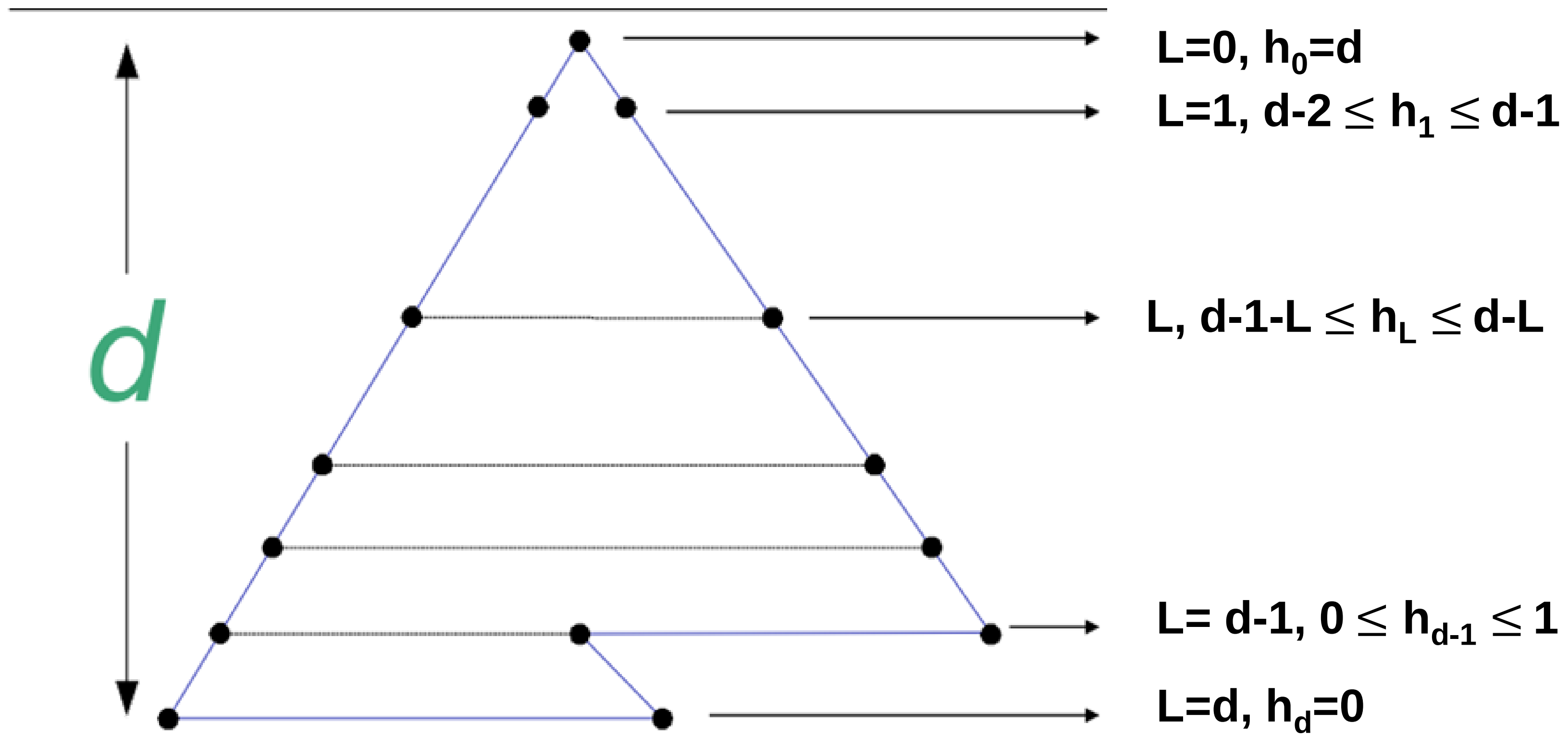
- **Work through the example on the board**
  A = {4, 1, 3, 2, 16, 9, 10, 14, 8, 7}



BUILDHEAP() RUNNING TIME

- Each call to `HEAPIFY()` takes $O(\log n)$ time
- There are $O(n)$ such calls ($\lfloor n/2 \rfloor$ calls indeed)
- Thus the running time is $O(n \log n)$
  - *Is this a correct asymptotic upper bound?*
  - *Is this an asymptotically tight bound?*
- A tighter bound is actually $O(n)$
  - *How can this be? Is there a flaw in the above reasoning?*

BUILDHEAP() : TIGHTER RUNNING TIME ANALYSIS



Let h_L denote height of a node at level L
We have $d-1-L \le h_L \le d-L$

BUILDHEAP() : TIGHTER RUNNING TIME ANALYSIS

- **Assume that all nodes at the last complete level (*l = d – 1*) are processed (upper bound)**

$$T(n) \le \sum_{l=0}^{d-1} n_l \; O(h_l) = O\left(\sum_{l=0}^{d-1} n_l \, h_l \right)$$

$$T(n) \le O\left(\sum_{l=0}^{d-1} 2^l \; (d - l)\right)$$

n_l = # of nodes at level $l \le 2^l$
h_l = height of nodes at level $l \le d-l$

Let $h = d - l \Rightarrow l = d - h$ (change of variables)

$$T(n) \le O\left(\sum_{h=1}^{d} h\, 2^{d-h} \right) = O\left(\sum_{h=1}^{d} h\, 2^d / 2^h \right) = O\left(2^d \sum_{h=1}^{d} h\, (1/2)^h \right)$$

But $2^d = \Theta(n) \Rightarrow T(n) \le O\left(n \sum_{h=1}^{d} h\, (1/2)^h \right)$

BUILDHEAP() : TIGHTER RUNNING TIME ANALYSIS

- $\sum_{h=\mathbf{1}}^{d} h\,(1/2)^h \le \sum_{h=\mathbf{0}}^{d} h\,(1/2)^h \le \sum_{h=\mathbf{0}}^{\infty} h\,(1/2)^h$
- **Recall infinite decreasing geometric series**
- $\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$ where |x| < 1
- **Differentiate both sides**
- $\sum_{k=0}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2}$
- **Then multiply both sides by *x***
- $\sum_{k=0}^{\infty} kx^{k} = \frac{x}{(1-x)^2}$

BUILD-HEAP: TIGHTER RUNNING TIME ANALYSIS

- $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$
- **In our case: *x=1/2* and *k=h***
- $\sum_{h=0}^{\infty} h\left(\frac{1}{2}\right)^h = \frac{1/2}{\left(1-\frac{1}{2}\right)^2} = 2$
- T(n) ≤ O(n $\sum_{h=1}^{d} h(1/2)^h$) = O(2n) = O(n)

BUILD-HEAP: TIGHTER RUNNING TIME ANALYSIS

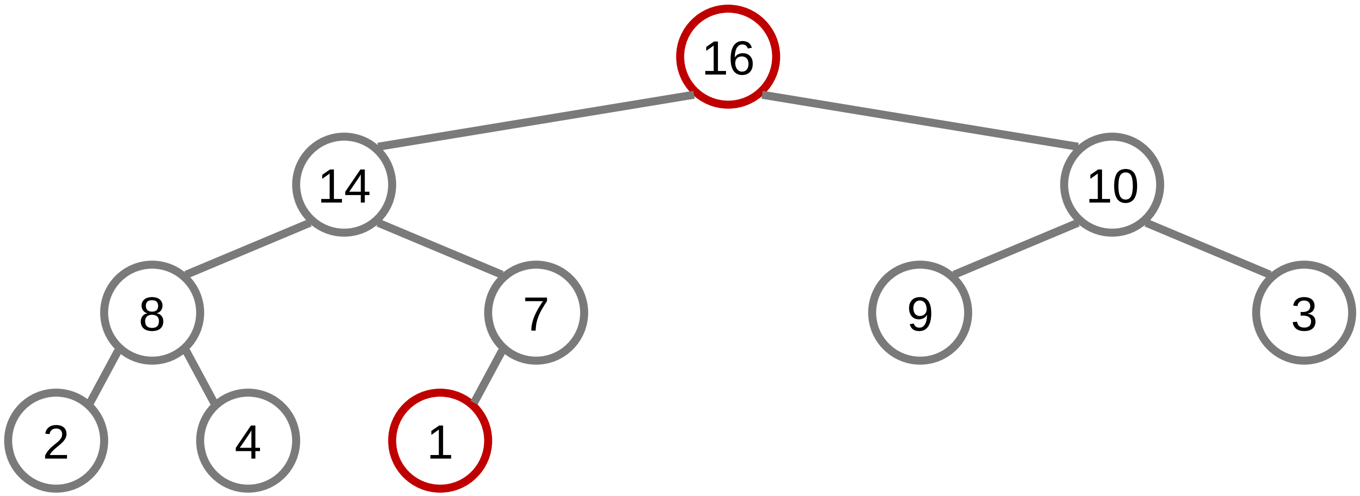
- $\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$
- **In our case: *x=1/2* and *k=h***
- $\sum_{h=0}^{\infty} h\left(\frac{1}{2}\right)^h = \frac{1/2}{\left(1-\frac{1}{2}\right)^2} = 2$
- T(n) ≤ O(n $\sum_{h=1}^{d} h(1/2)^h$) = O(2n) = O(n)

- **Intuition:**
  - Most HEAPIFY() calls occur at lower levels, since most of the nodes in a tree are at lower levels.
  - Those calls are very fast, O(1) for the lowest levels that contain most of the nodes.
  - Only relatively few nodes at upper levels require O(log n) HEAPIFY() cost.

HEAPSORT

```
HEAPSORT (A, n)
        BUILDHEAP(A, n)
        Repeat until n = 2
                //The largest element is the root, which
                should be the last element in the sorted
                array
                swap A[1] ↔ A[n]
                //Discard node n from the heap (reduce
                heap size)
                //Sub-trees rooted at children of the root
                are heaps but the new root may violate
                heap property
                HEAPIFY(A, n - 1)
                set n = n - 1
```

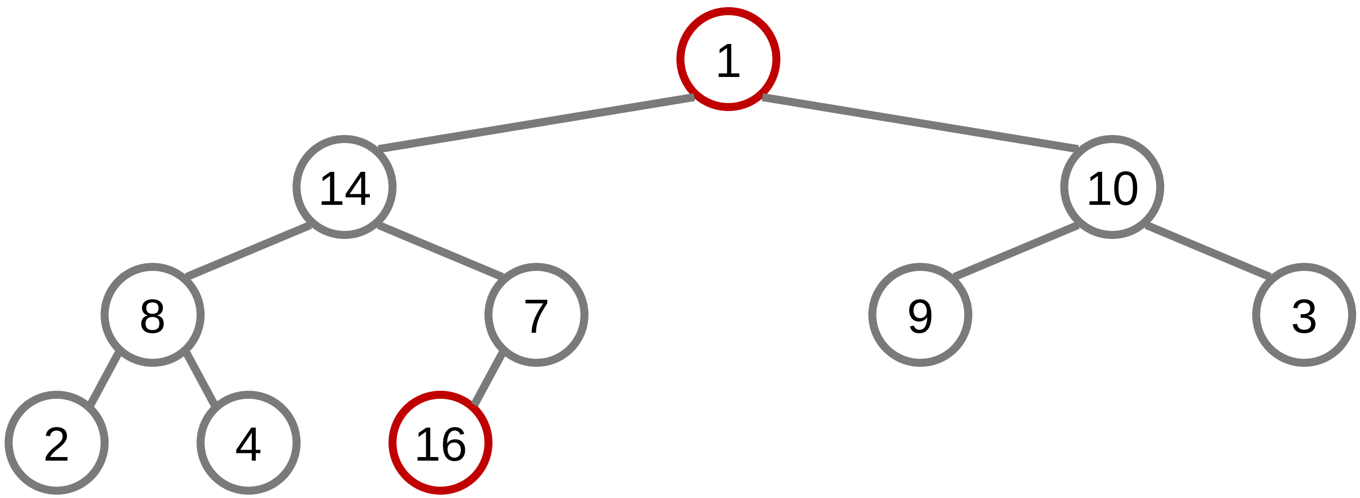
HEAPSORT EXAMPLE



A =

16	14	10	8	7	9	3	2	4	1

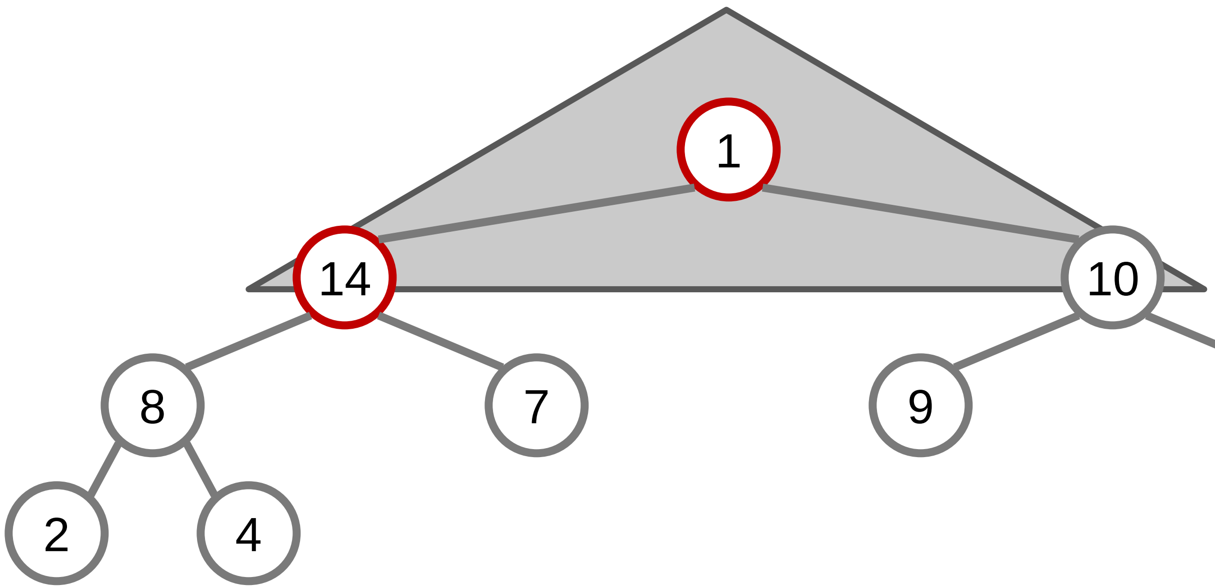
HEAPSORT EXAMPLE



A =

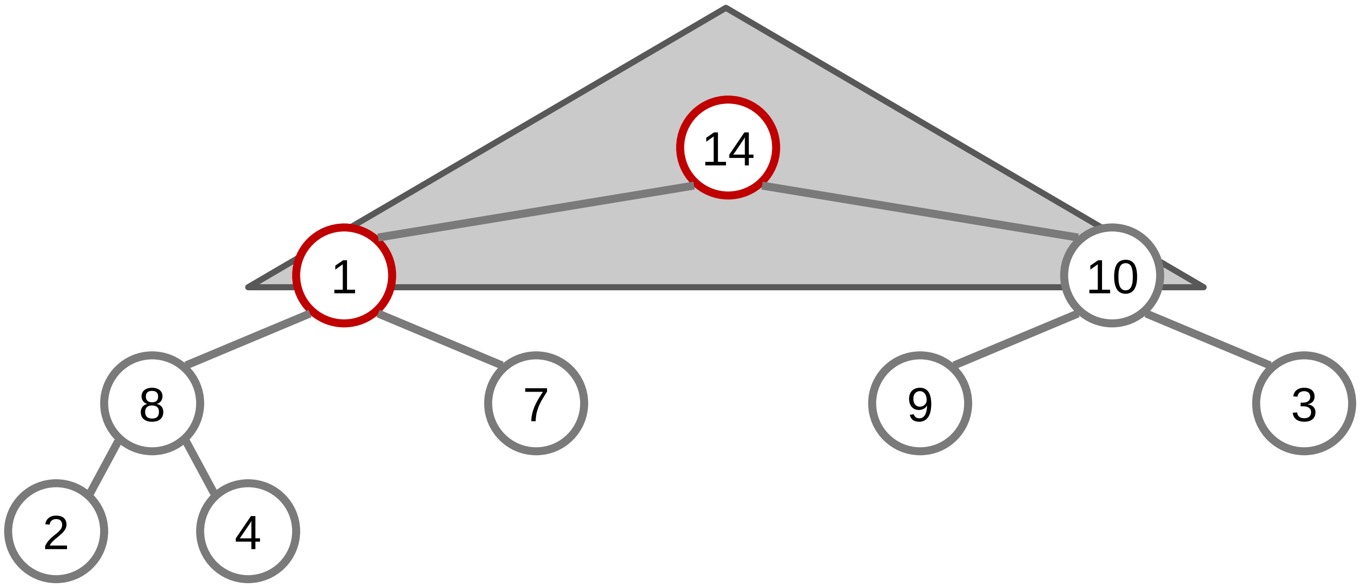
1	14	10	8	7	9	3	2	4	16

HEAPSORT EXAMPLE



A =	1	14	10	8	7	9	3	2	4	16

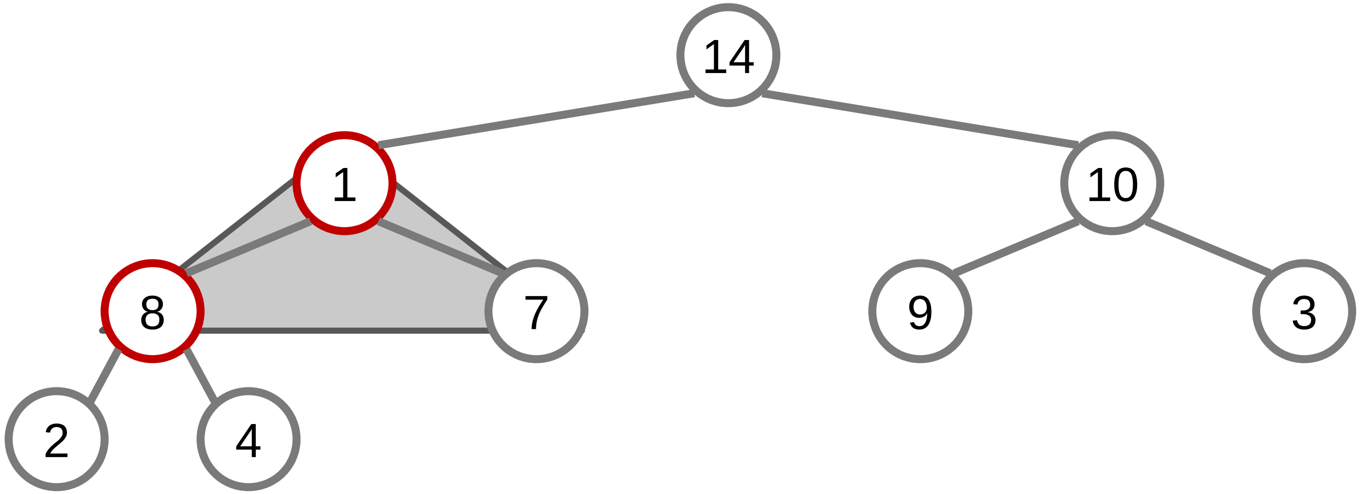
HEAPSORT EXAMPLE



A =

14	1	10	8	7	9	3	2	4	16

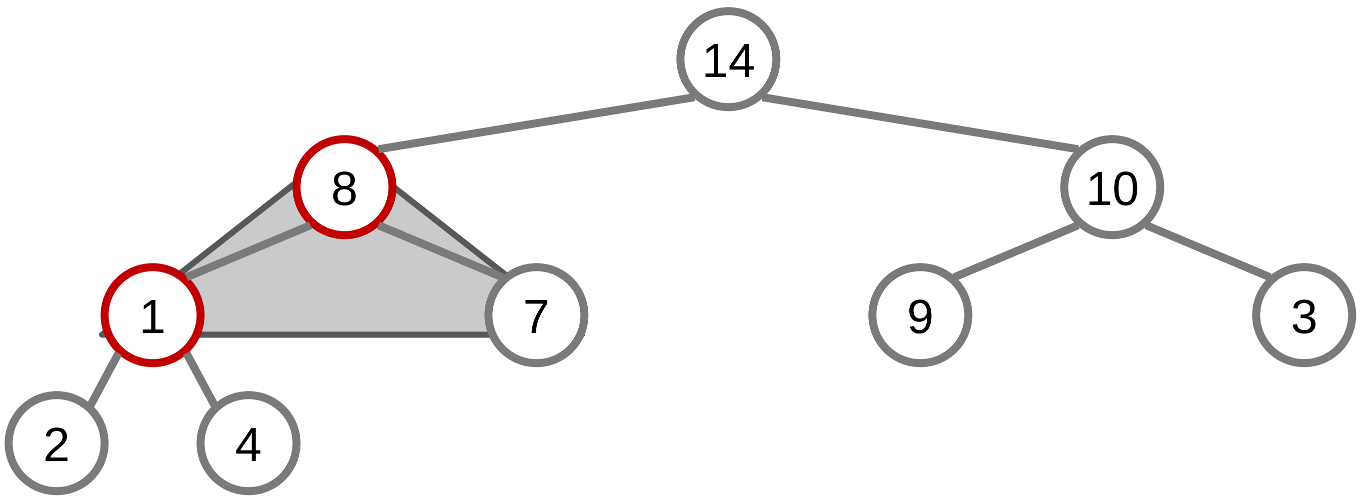
HEAPSORT EXAMPLE



A =

14	1	10	8	7	9	3	2	4	16

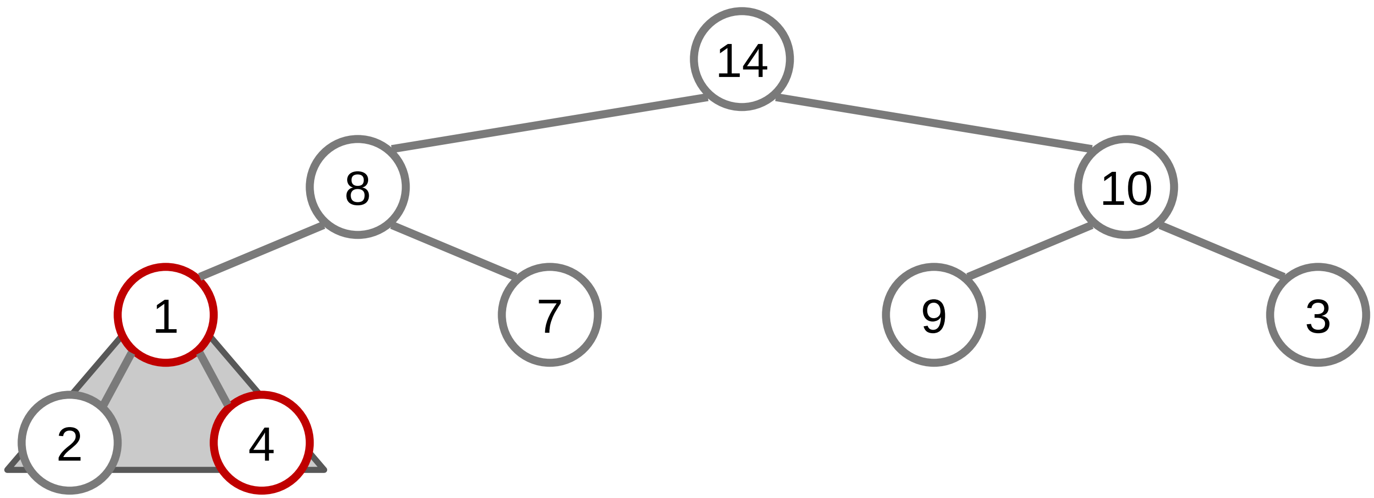
HEAPSORT EXAMPLE



A =

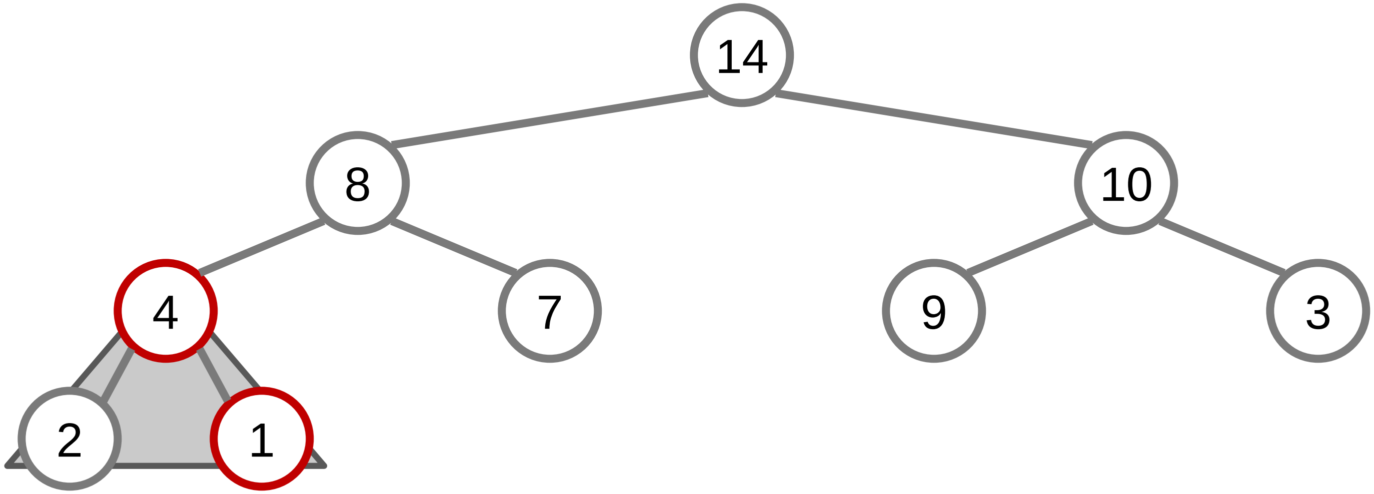
14	8	10	1	7	9	3	2	4	16

HEAPSORT EXAMPLE



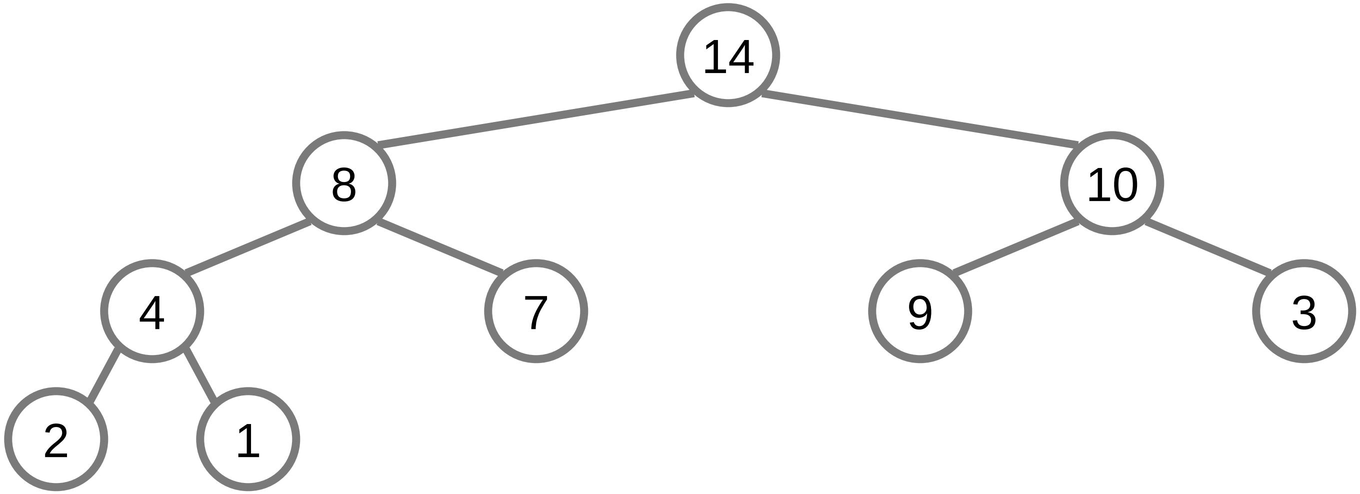
A =									
14	8	10	1	7	9	3	2	4	16

HEAPSORT EXAMPLE



A =	14	8	10	4	7	9	3	2	1	16

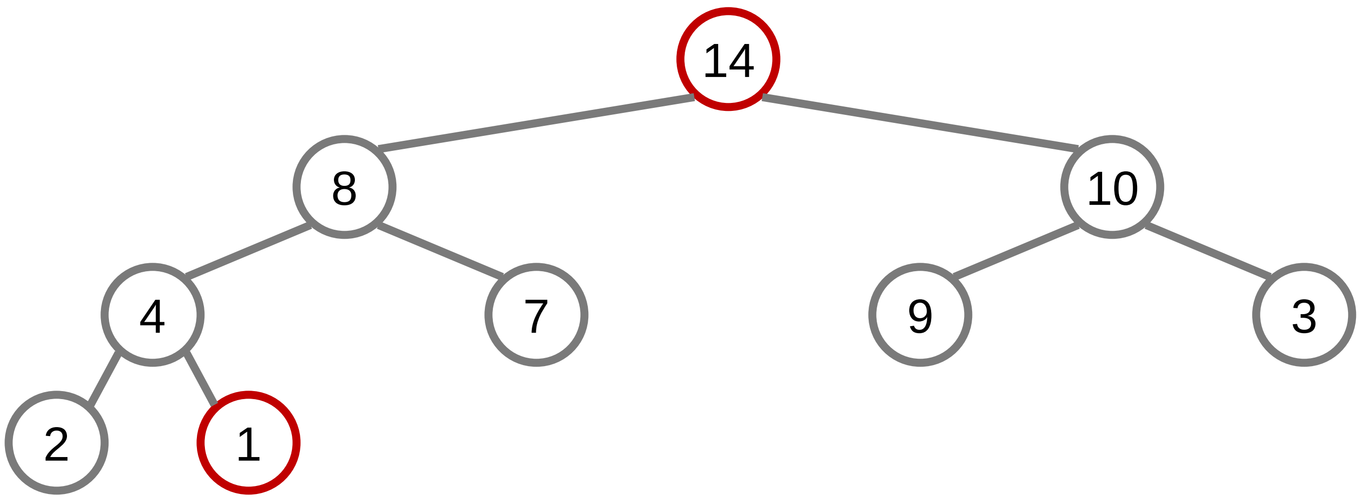
HEAPSORT EXAMPLE



A =

14	8	10	4	7	9	3	2	1	16

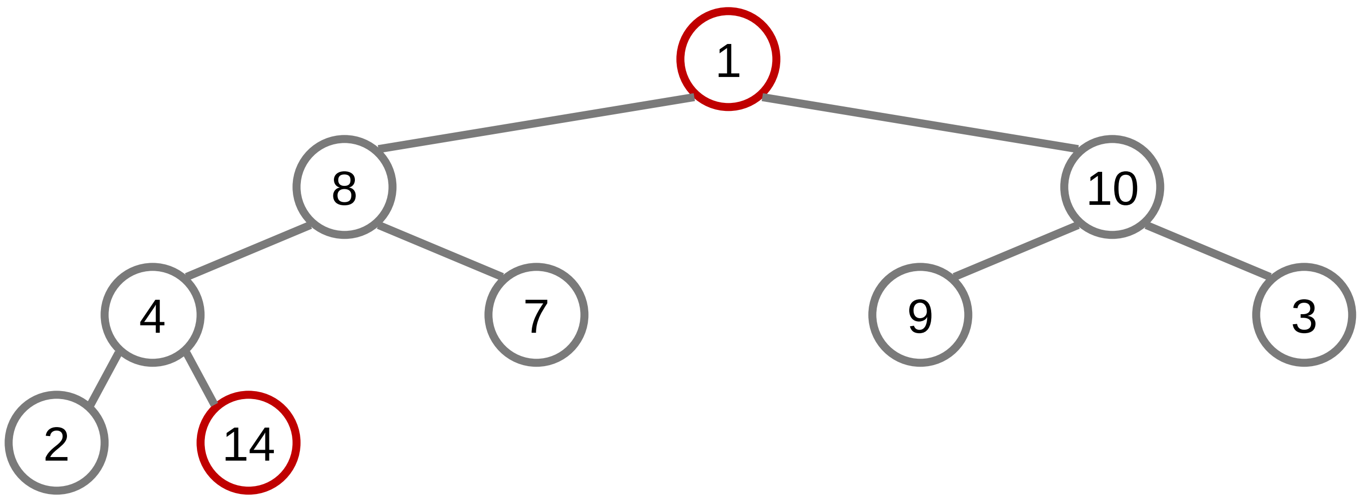
HEAPSORT EXAMPLE



A =

14	8	10	4	7	9	3	2	1	16

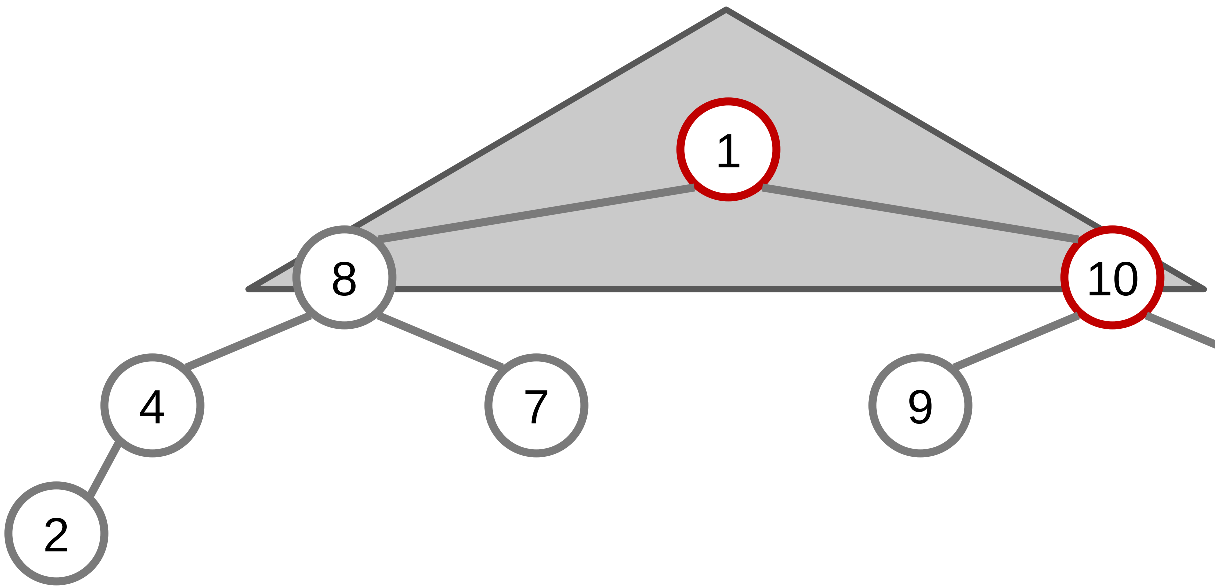
HEAPSORT EXAMPLE



A =

1	8	10	4	7	9	3	2	14	16

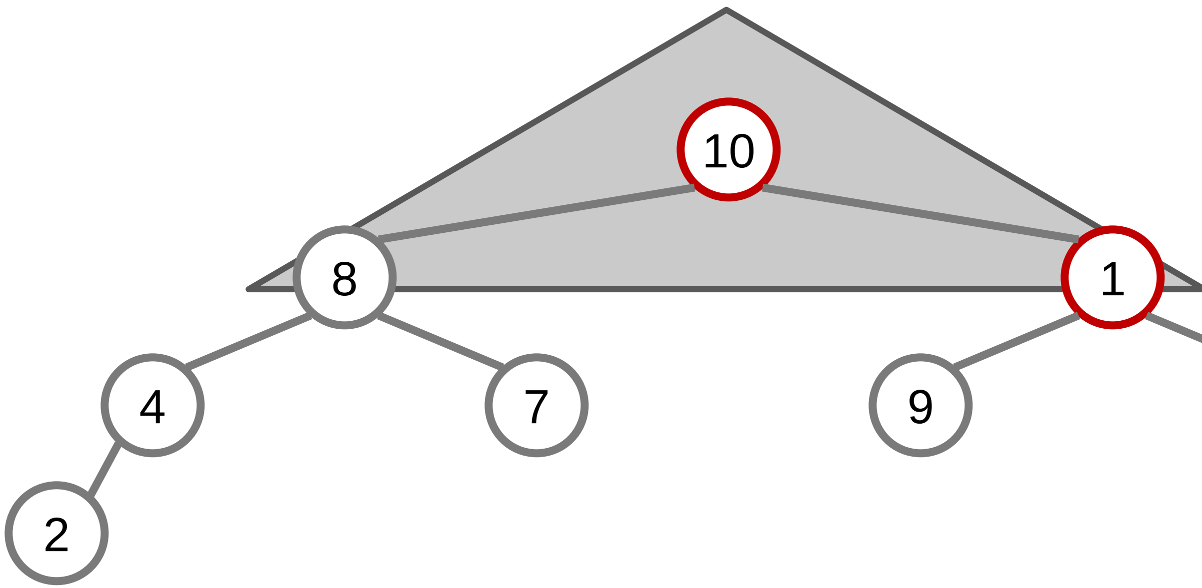
HEAPSORT EXAMPLE



A =

1	8	10	4	7	9	3	2	14	16

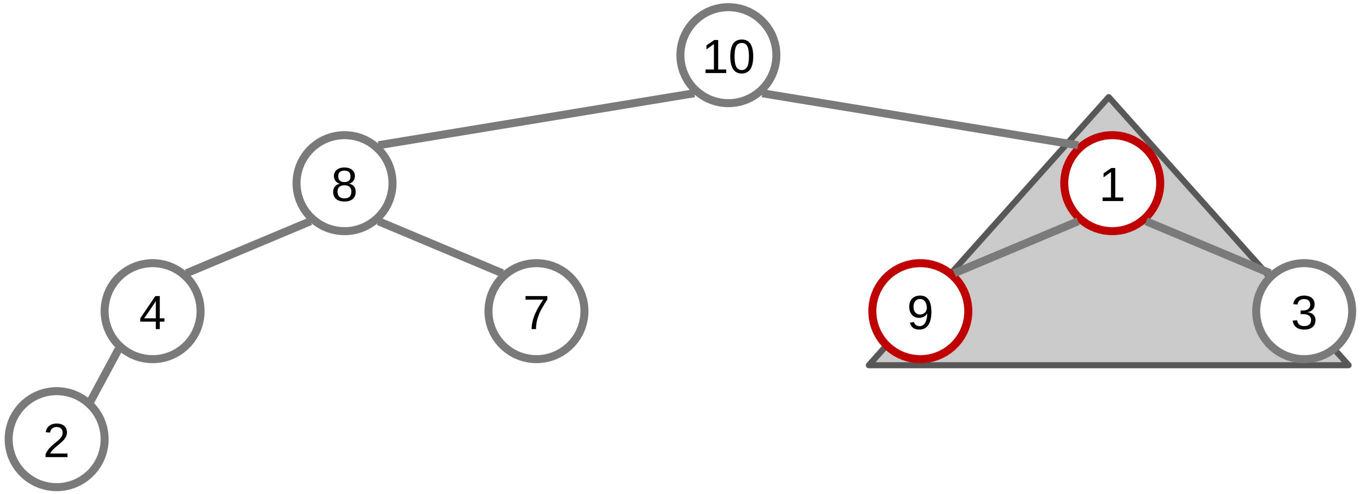
HEAPSORT EXAMPLE



A =

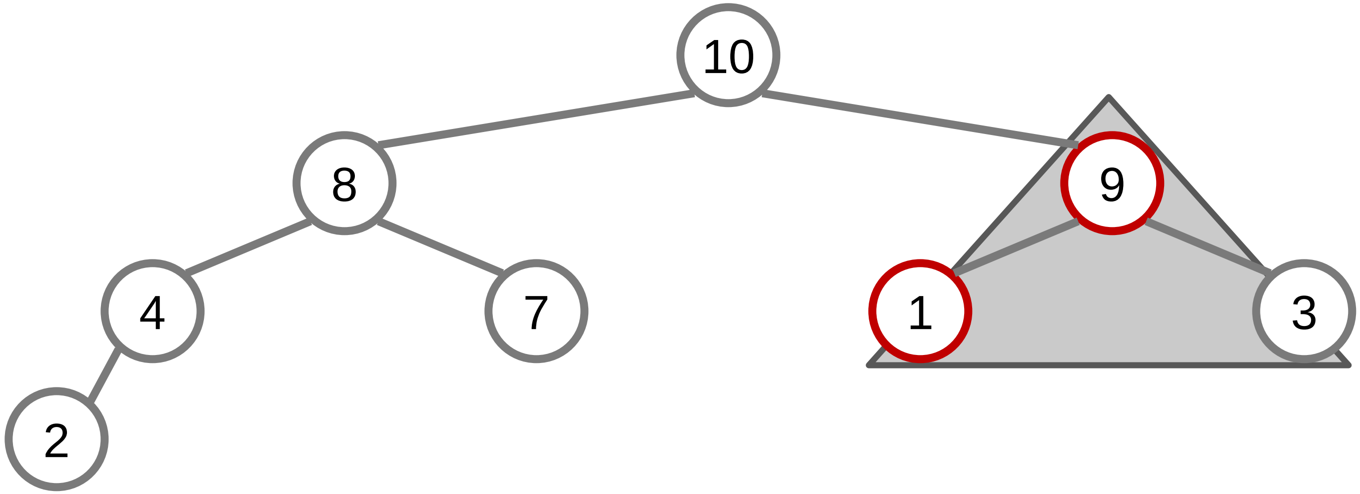
10	8	1	4	7	9	3	2	14	16

HEAPSORT EXAMPLE



A =	10	8	1	4	7	9	3	2	14	16

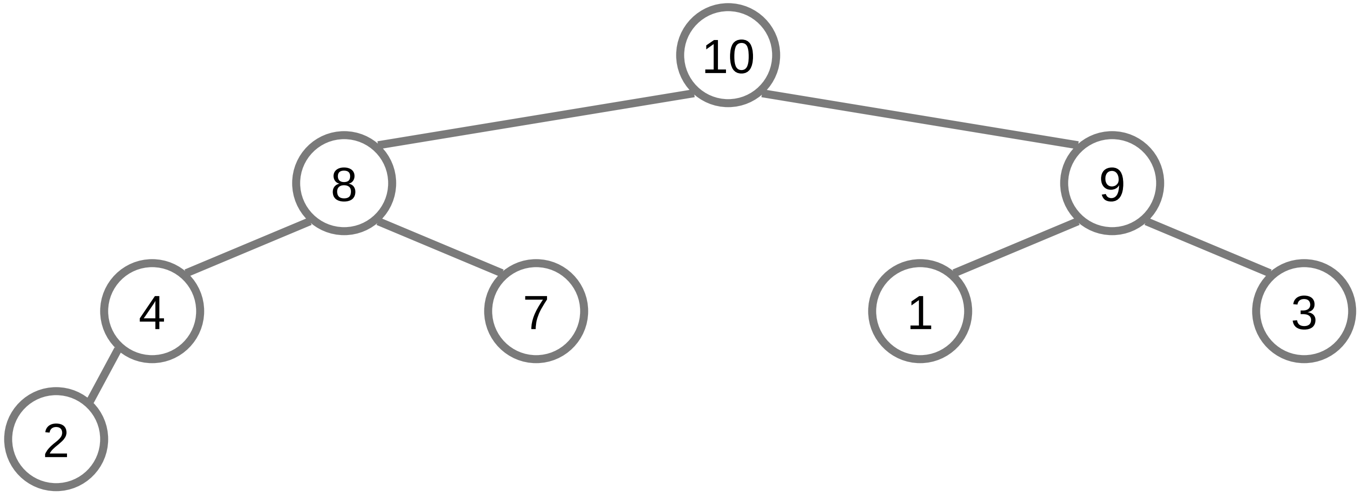
HEAPSORT EXAMPLE



A =

10	8	9	4	7	1	3	2	14	16

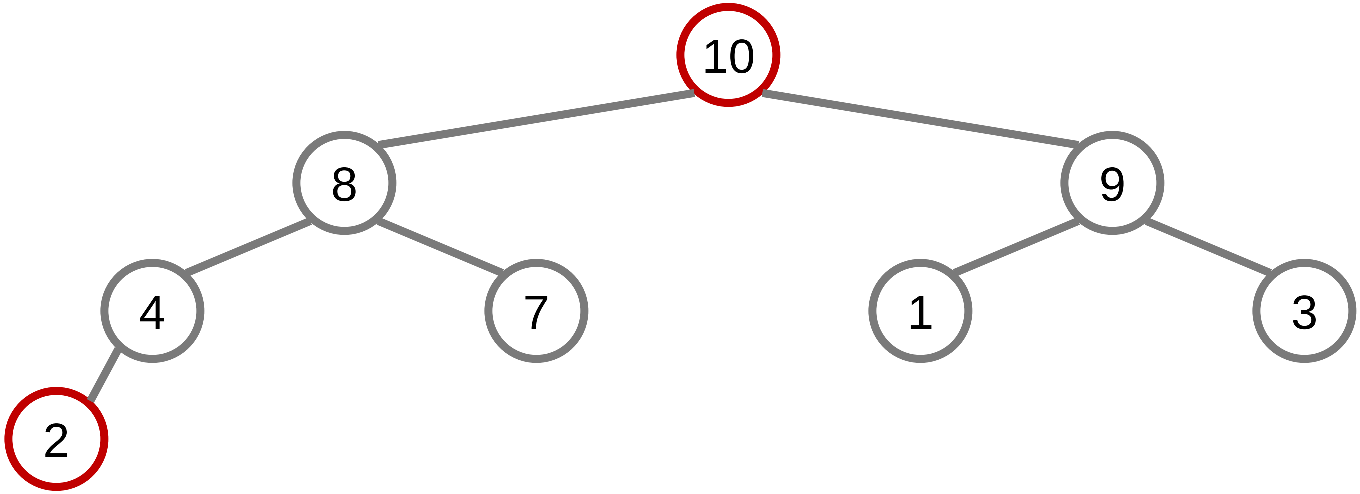
HEAPSORT EXAMPLE



A =

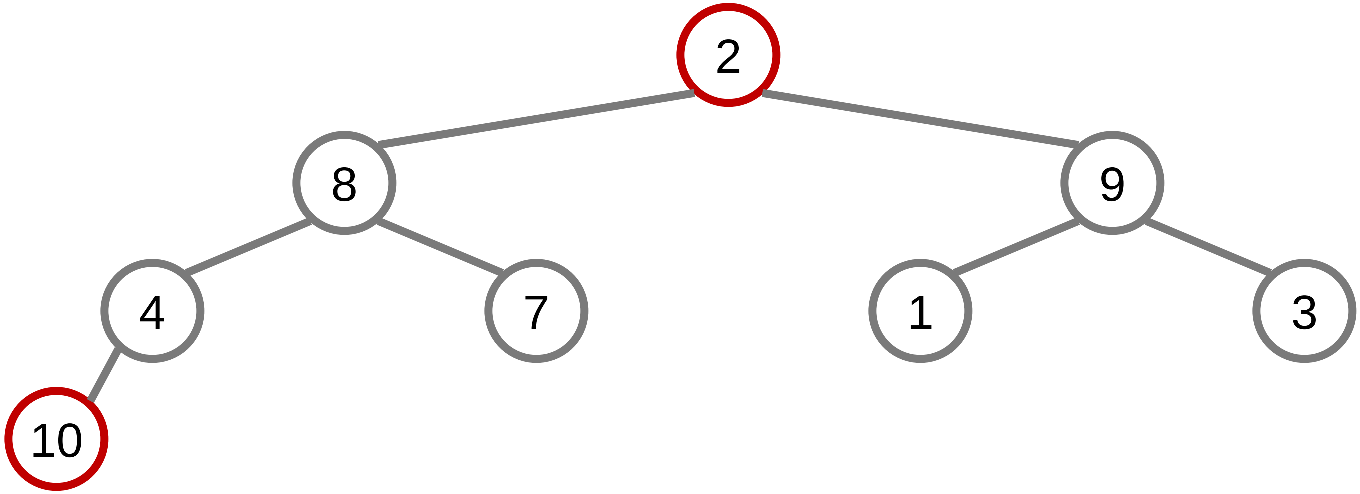
10	8	9	4	7	1	3	2	14	16

HEAPSORT EXAMPLE



A =	10	8	9	4	7	1	3	2	14	16

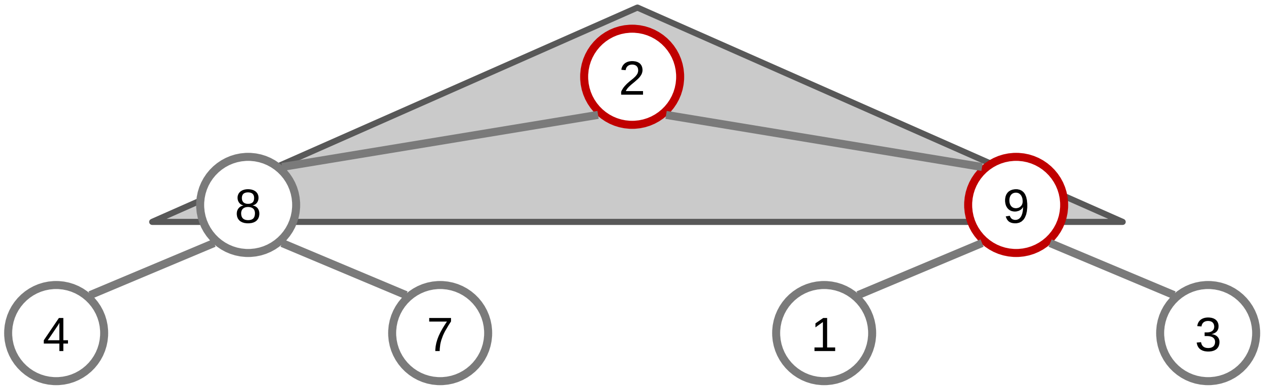
HEAPSORT EXAMPLE



A =

2	8	9	4	7	1	3	10	14	16

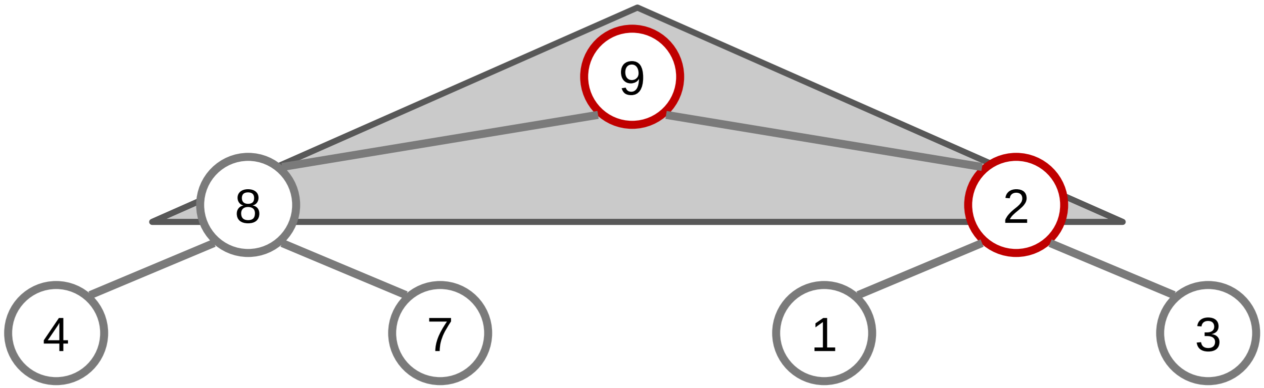
HEAPSORT EXAMPLE



A =

2	8	9	4	7	1	3	10	14	16

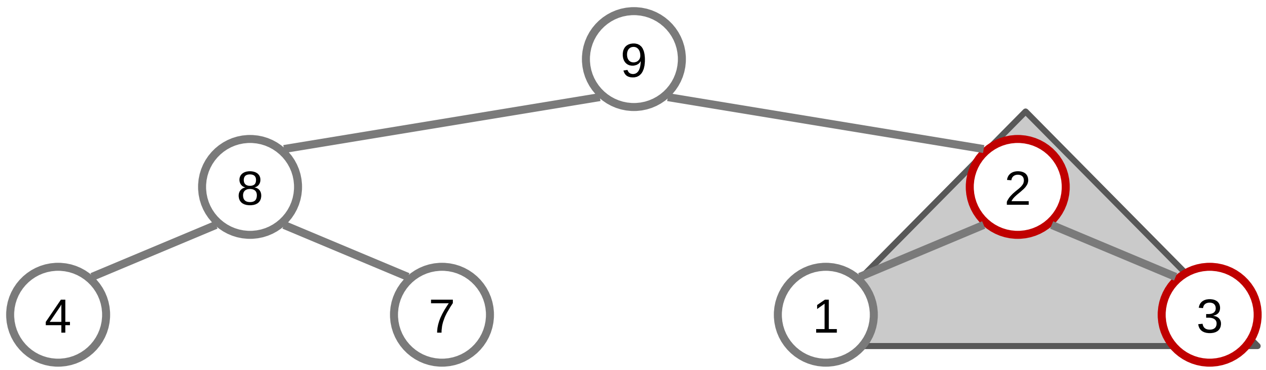
HEAPSORT EXAMPLE



A =

9	8	2	4	7	1	3	10	14	16

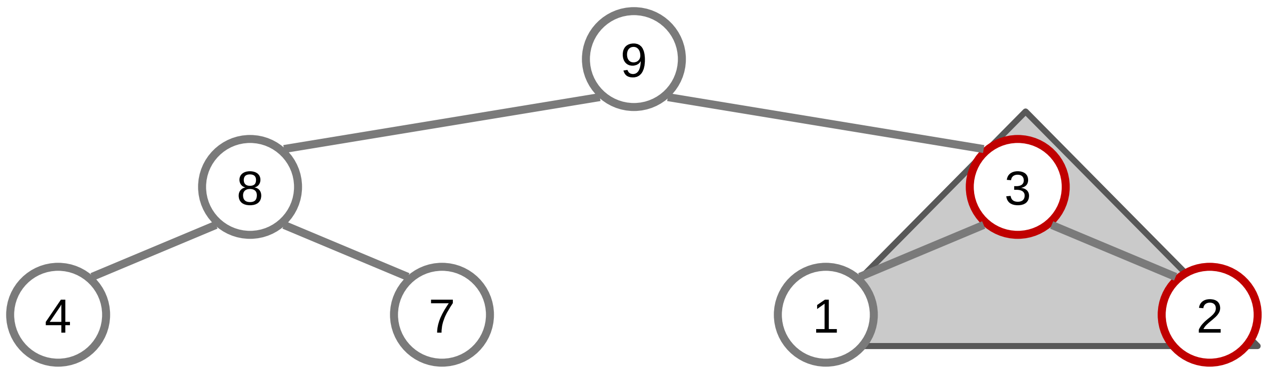
HEAPSORT EXAMPLE



A =

9	8	2	4	7	1	3	10	14	16

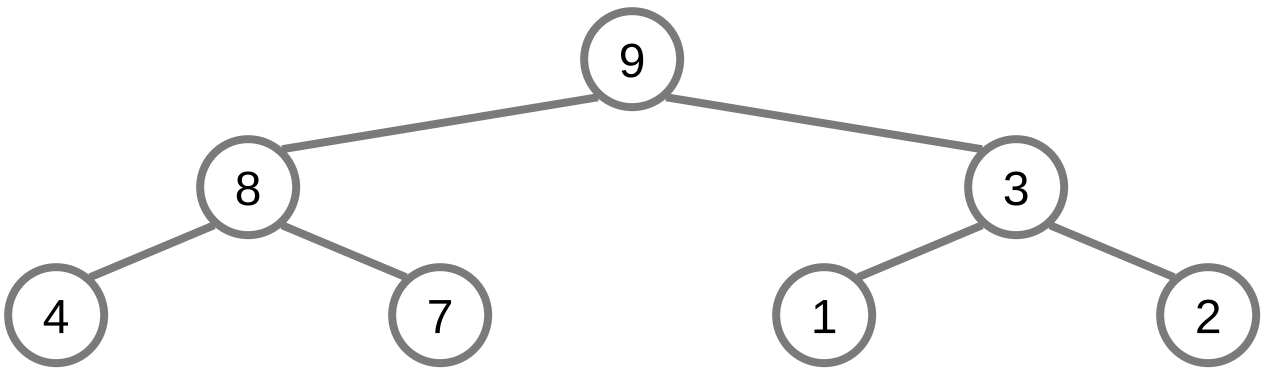
HEAPSORT EXAMPLE



A =

9	8	3	4	7	1	2	10	14	16

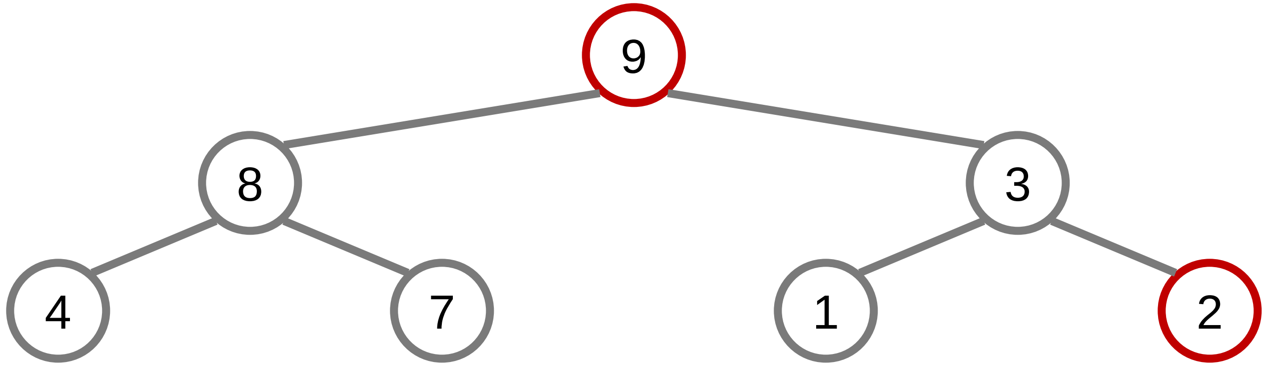
HEAPSORT EXAMPLE



A =

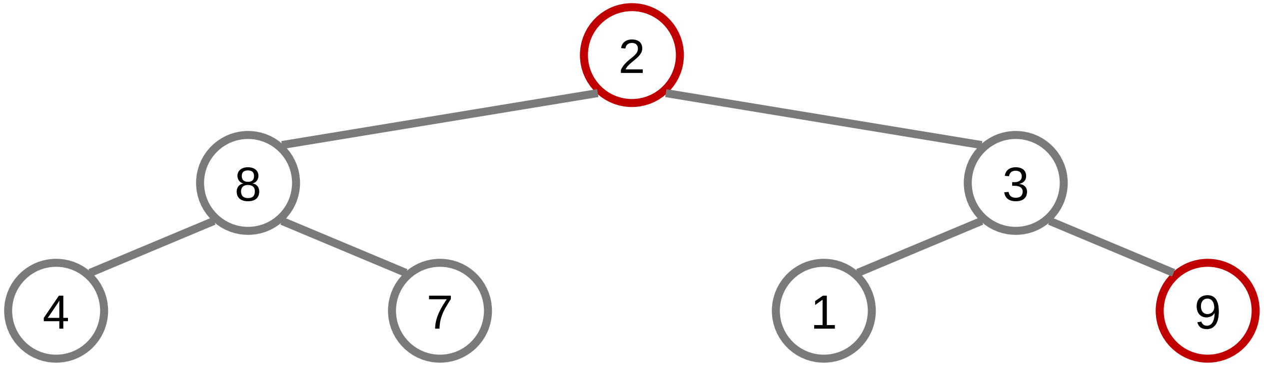
9	8	3	4	7	1	2	10	14	16

HEAPSORT EXAMPLE



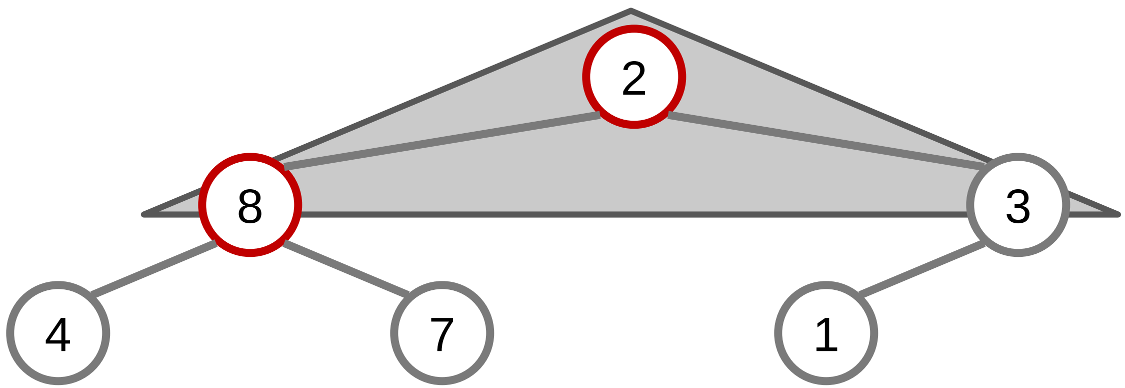
A =	9	8	3	4	7	1	2	10	14	16

HEAPSORT EXAMPLE



A =	2	8	3	4	7	1	9	10	14	16

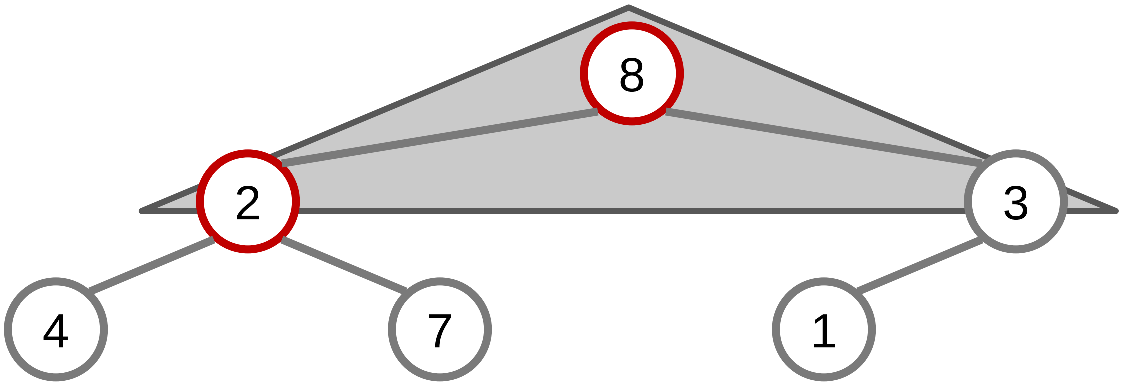
HEAPSORT EXAMPLE



A =

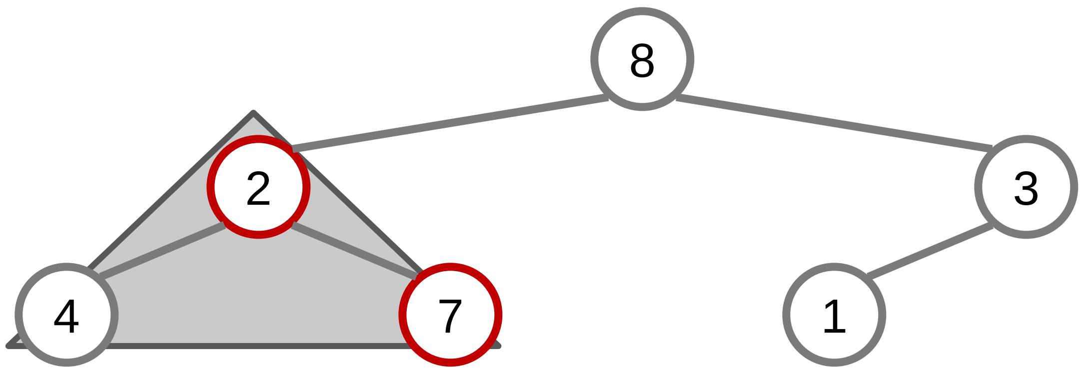
2	8	3	4	7	1	9	10	14	16

HEAPSORT EXAMPLE



A =	8	2	3	4	7	1	9	10	14	16

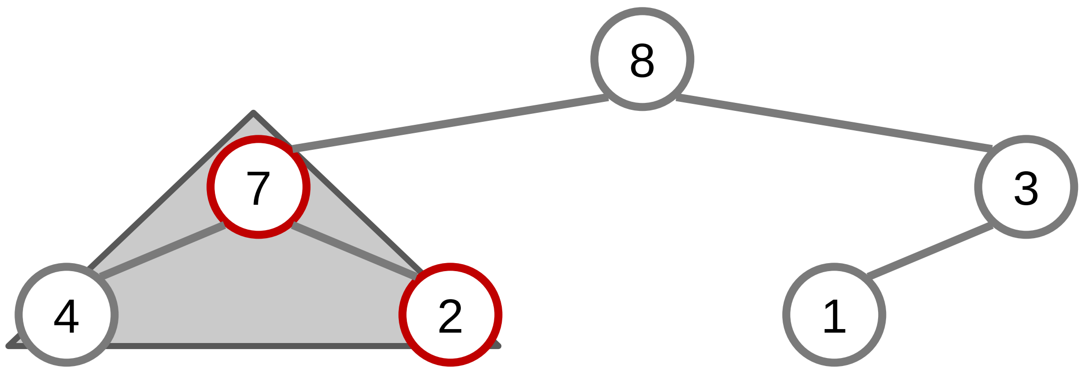
HEAPSORT EXAMPLE



A =

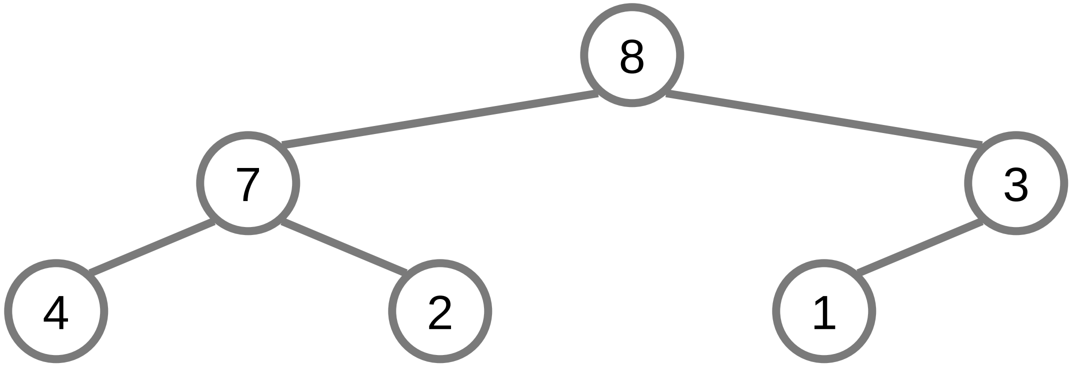
8	2	3	4	7	1	9	10	14	16

HEAPSORT EXAMPLE



A =	8	7	3	4	2	1	9	10	14	16

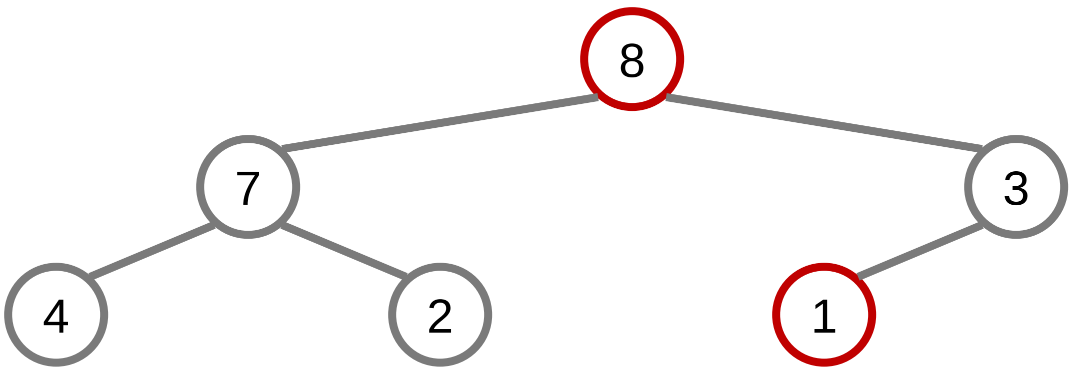
HEAPSORT EXAMPLE



A =

8	7	3	4	2	1	9	10	14	16

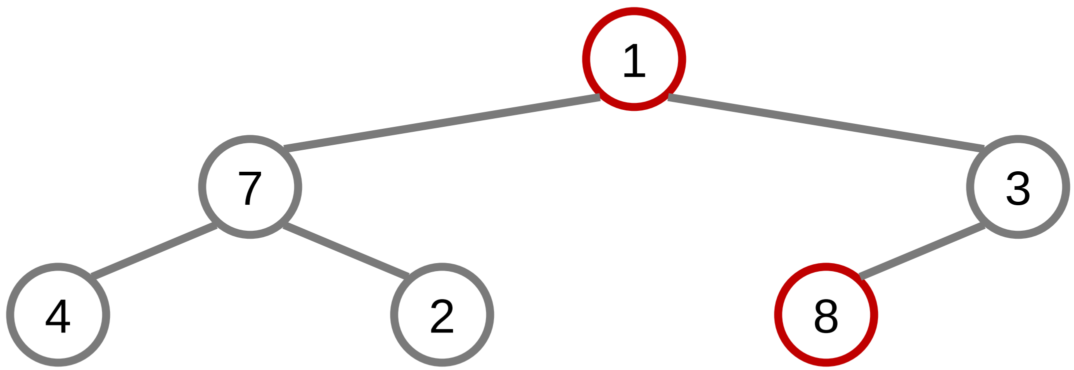
HEAPSORT EXAMPLE



A =

8	7	3	4	2	1	9	10	14	16

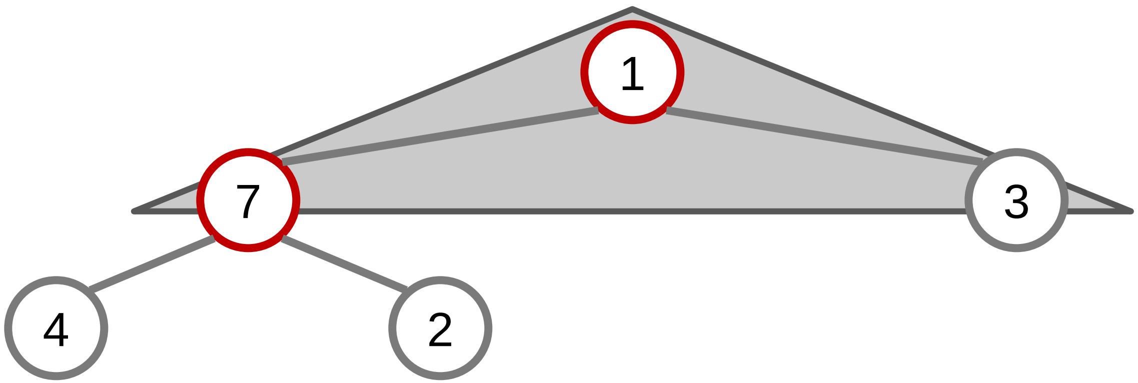
HEAPSORT EXAMPLE



A =

1	7	3	4	2	8	9	10	14	16

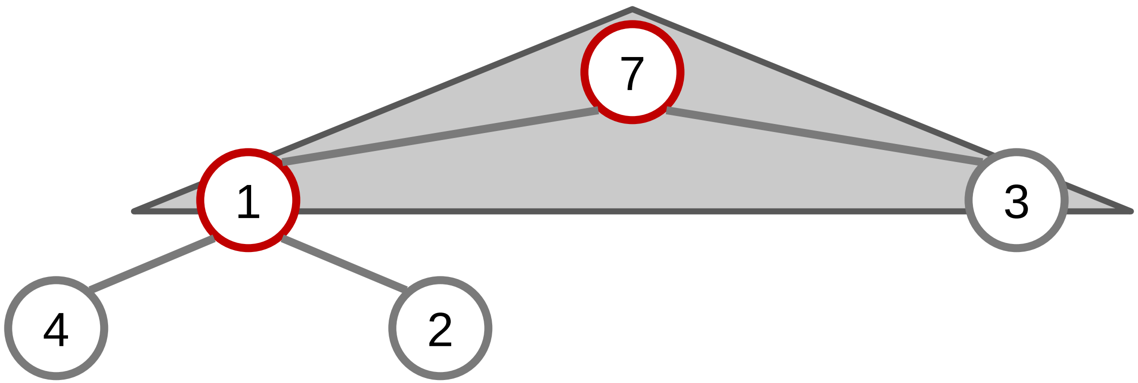
HEAPSORT EXAMPLE



A =

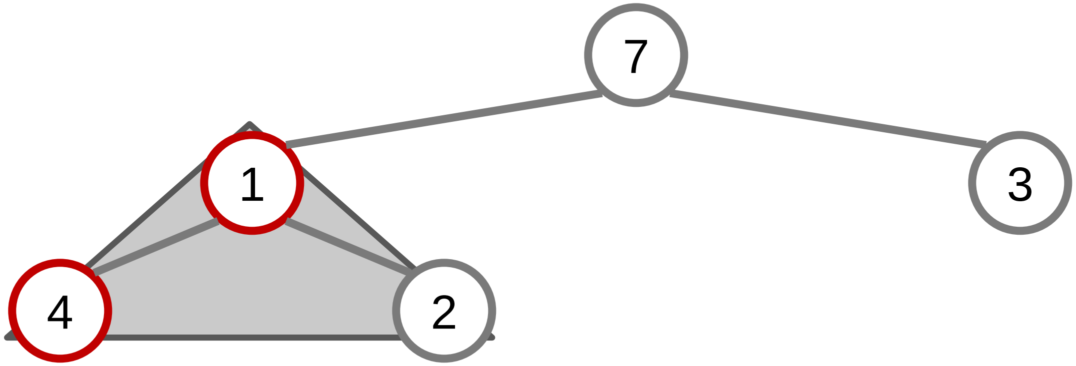
1	7	3	4	2	8	9	10	14	16

HEAPSORT EXAMPLE



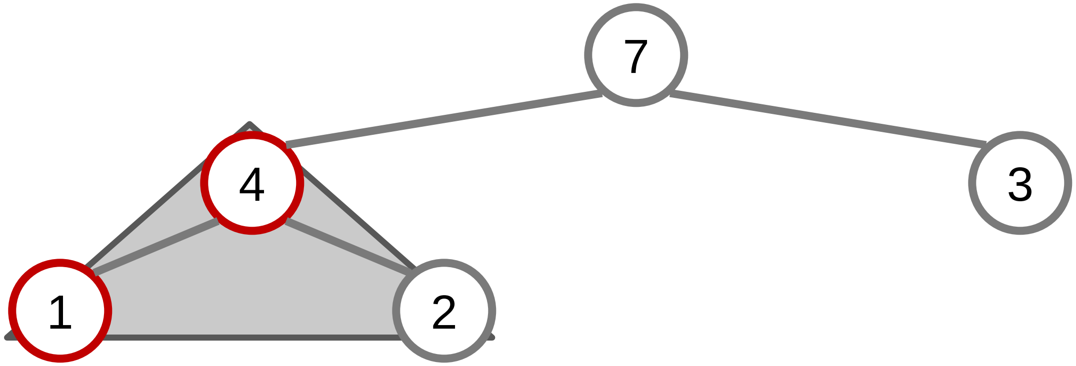
A =										
	7	1	3	4	2	8	9	10	14	16

HEAPSORT EXAMPLE



A =	7	1	3	4	2	8	9	10	14	16

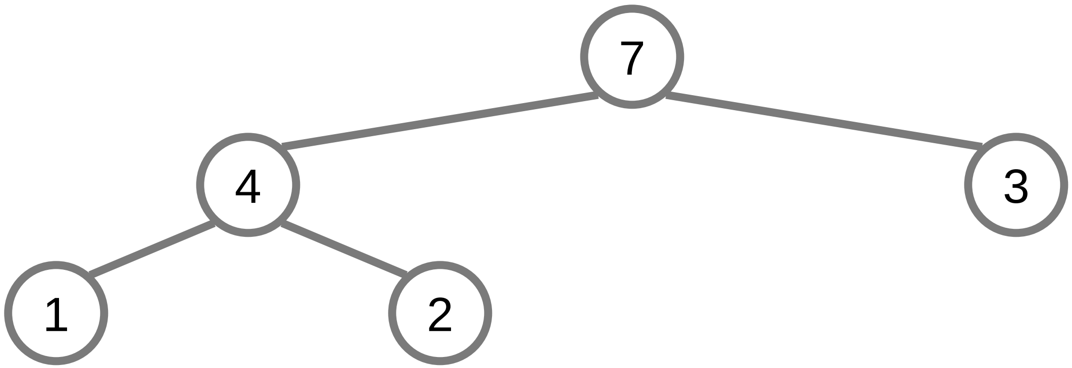
HEAPSORT EXAMPLE



A =

7	4	3	1	2	8	9	10	14	16

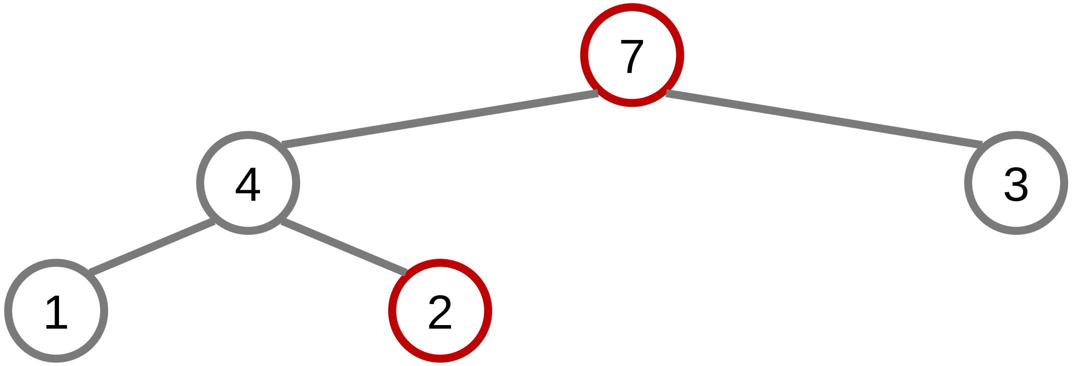
HEAPSORT EXAMPLE



A =

7	4	3	1	2	8	9	10	14	16

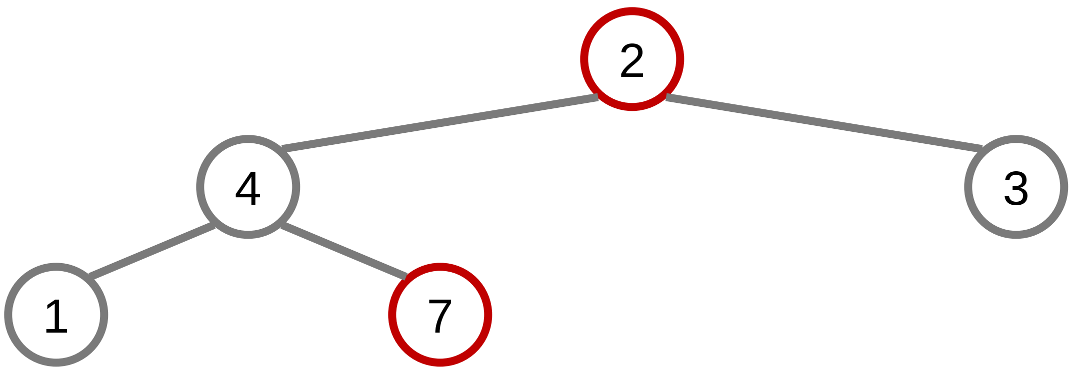
HEAPSORT EXAMPLE



A =

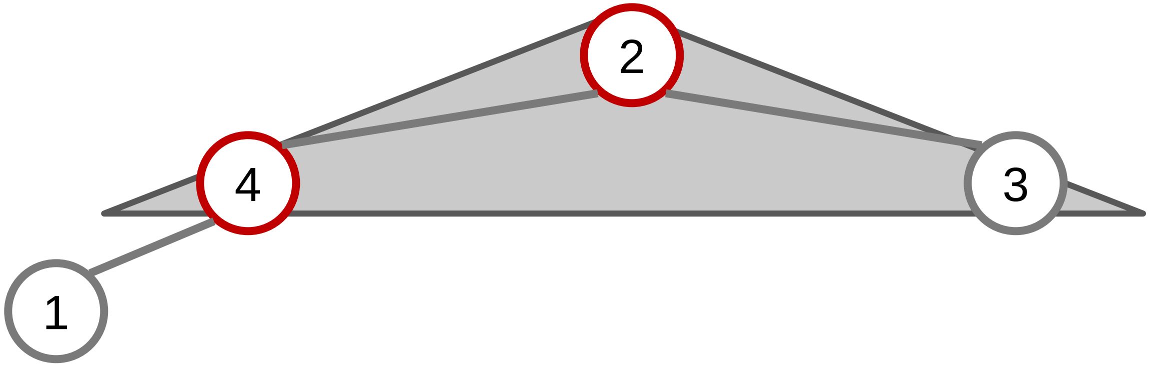
7	4	3	1	2	8	9	10	14	16

HEAPSORT EXAMPLE



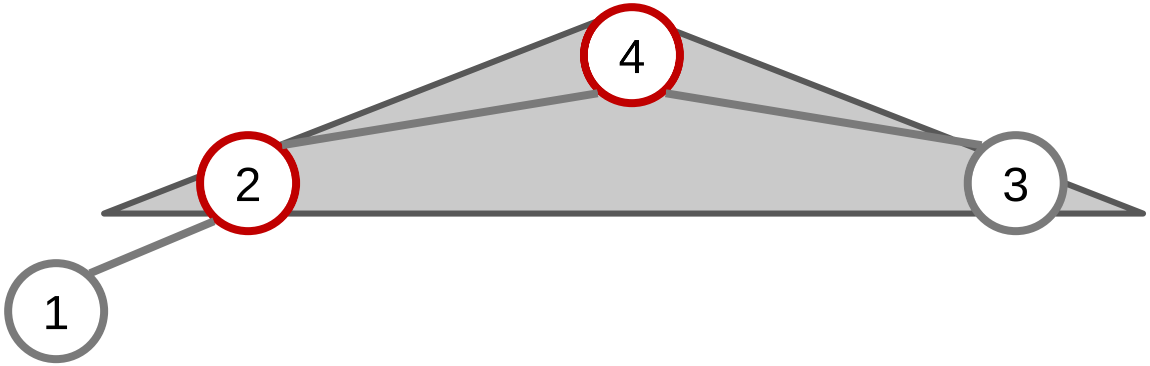
A =	2	4	3	1	7	8	9	10	14	16

HEAPSORT EXAMPLE



A =	2	4	3	1	7	8	9	10	14	16

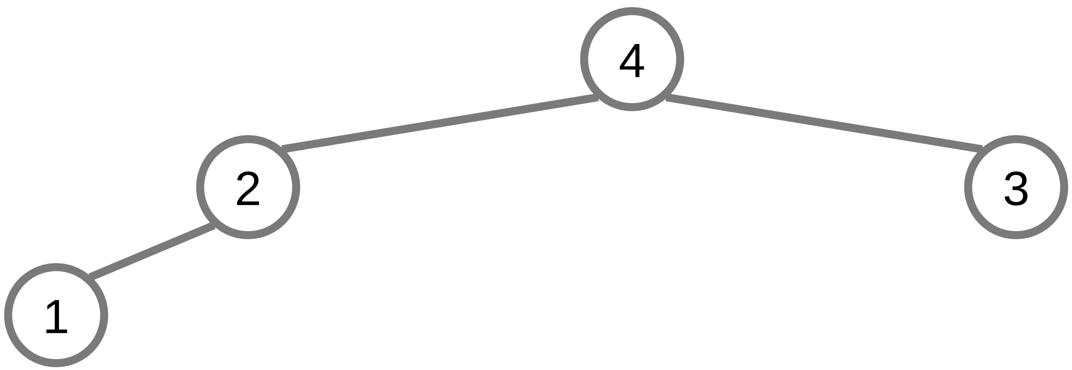
HEAPSORT EXAMPLE



A =

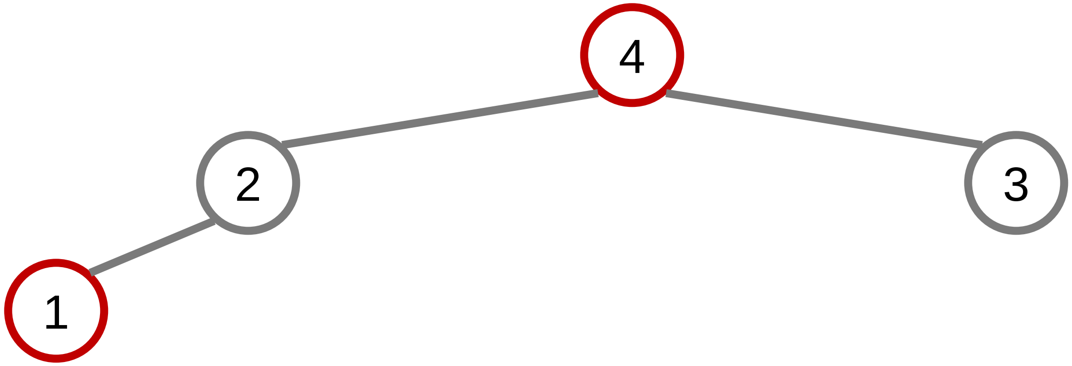
4	2	3	1	7	8	9	10	14	16

HEAPSORT EXAMPLE



A =	4	2	3	1	7	8	9	10	14	16

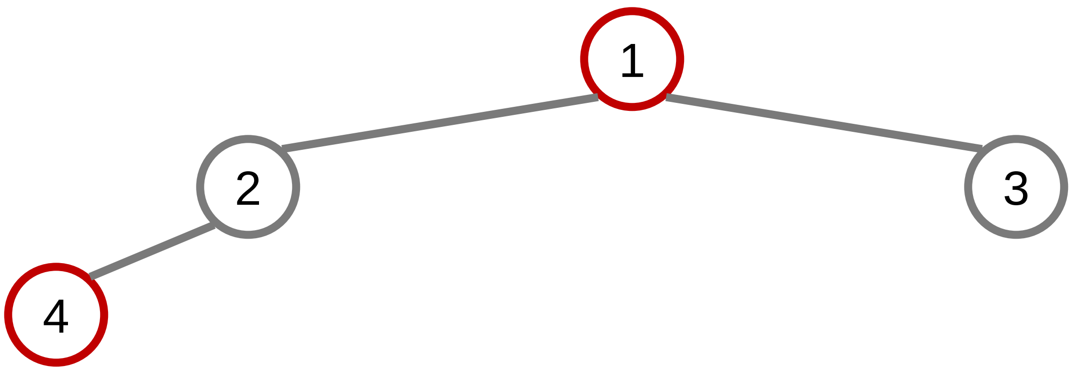
HEAPSORT EXAMPLE



A =

4	2	3	1	7	8	9	10	14	16

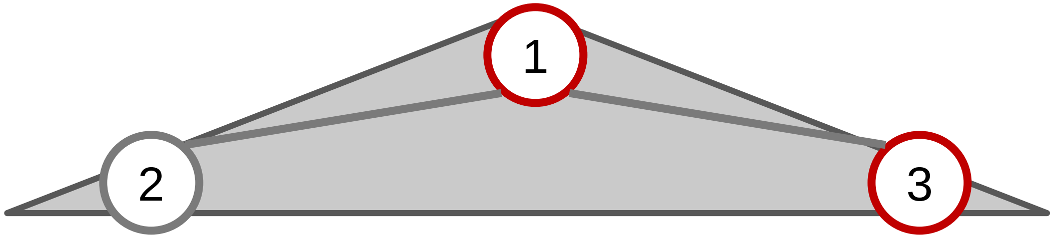
HEAPSORT EXAMPLE



A =

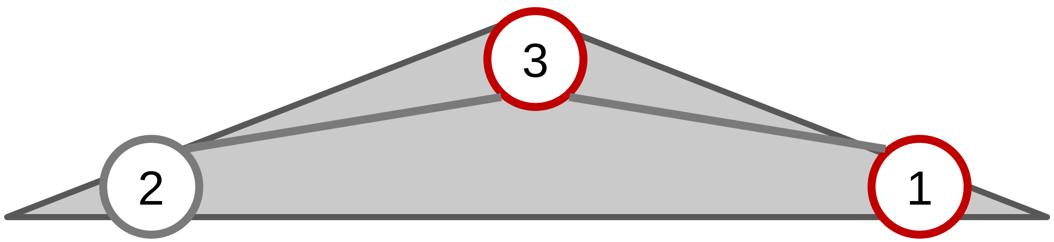
1	2	3	4	7	8	9	10	14	16

HEAPSORT EXAMPLE



A =	1	2	3	4	7	8	9	10	14	16

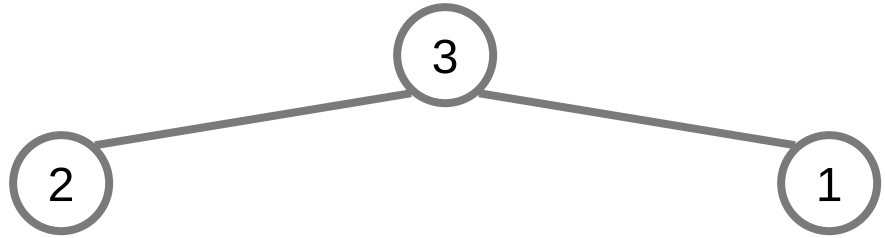
HEAPSORT EXAMPLE



A =

3	2	1	4	7	8	9	10	14	16

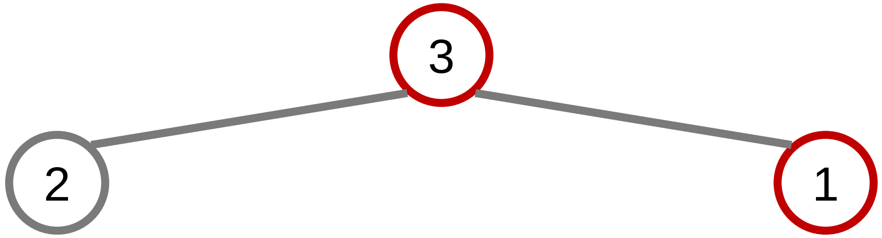
HEAPSORT EXAMPLE



A =

3	2	1	4	7	8	9	10	14	16

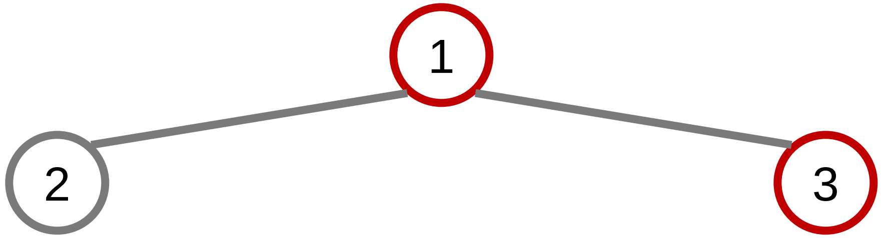
HEAPSORT EXAMPLE



A =

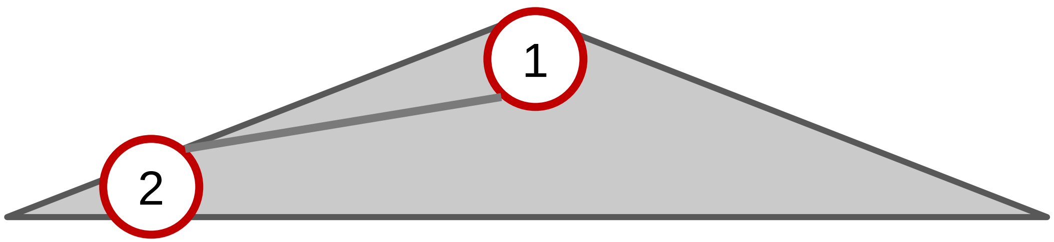
3	2	1	4	7	8	9	10	14	16

HEAPSORT EXAMPLE



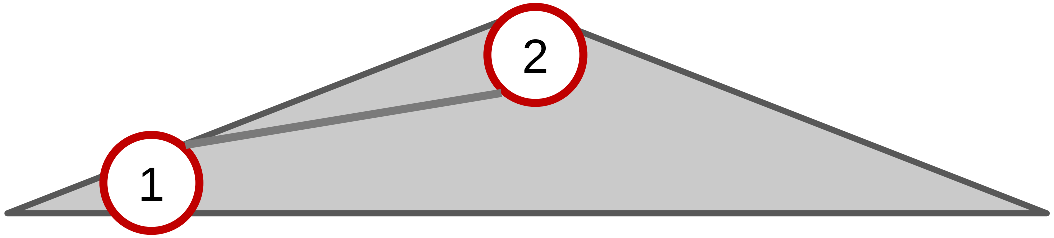
A =	1	2	3	4	7	8	9	10	14	16

HEAPSORT EXAMPLE



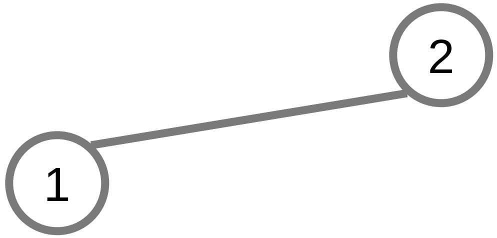
A =	1	2	3	4	7	8	9	10	14	16

HEAPSORT EXAMPLE



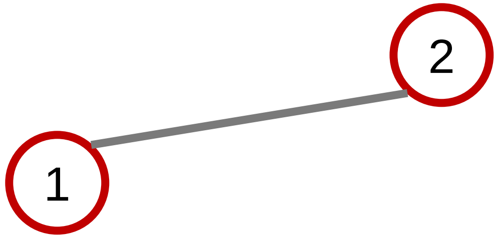
A =	2	1	3	4	7	8	9	10	14	16

HEAPSORT EXAMPLE



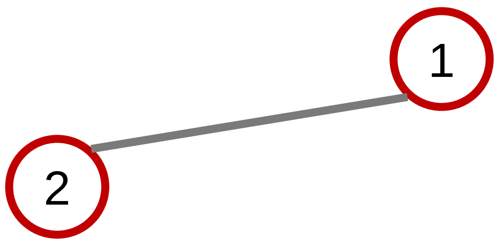
A =	2	1	3	4	7	8	9	10	14	16

HEAPSORT EXAMPLE



A =	2	1	3	4	7	8	9	10	14	16

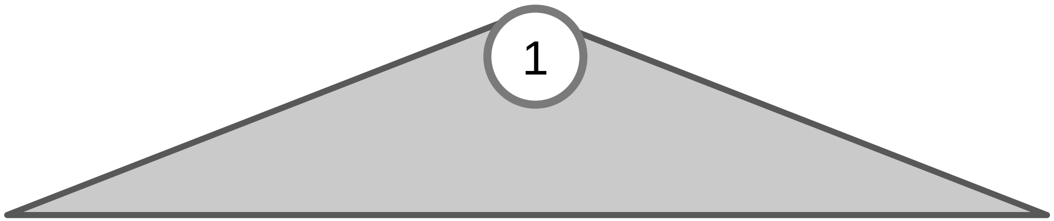
HEAPSORT EXAMPLE



A =

1	2	3	4	7	8	9	10	14	16

HEAPSORT EXAMPLE



A =

1	2	3	4	7	8	9	10	14	16

HEAPSORT EXAMPLE

A =	1	2	3	4	7	8	9	10	14	16

CONCLUSION

- **Heapsort is a very neat and clean algorithm**
- **Quicksort is faster in practice**
- **Why deal with Heapsort?**
  - Shows how to employ data structures to achieve more complicated functionality.
  - The **BUILDHEAP()** analysis is really important, since it does not result in the obvious guess!
  - Heaps are used in implementing Priority Queues
    - Which are used in game engines and operating systems for scheduling purposes. Read your book for more.

SORTING SO FAR

- **Insertion sort:**
  - Easy to code
  - Fast on small inputs (less than ~30 elements)
  - In-place
  - O(n) best case (nearly-sorted inputs)
  - $O(n^2)$ worst case (reverse-sorted inputs)
  - $O(n^2)$ average case (assuming all inputs are equally-likely)

SORTING SO FAR

- **Merge sort:**
  - Divide-and-conquer:
    - Split array in half
    - Recursively sort sub-arrays
    - Linear-time merge step
  - O(n log n) worst case, best case, and average case
  - Not in-place

SORTING SO FAR

- **Heap sort:**
  - Uses the very useful heap data structure
    - Nearly-complete binary tree
    - Heap property:
      - parent key ≥ children's keys (max-heap)
      - parent key ≤ children's keys (min-heap)
  - O(n log n) worst case, best case, average case
  - In-place
  - Many swap operations

SORTING SO FAR

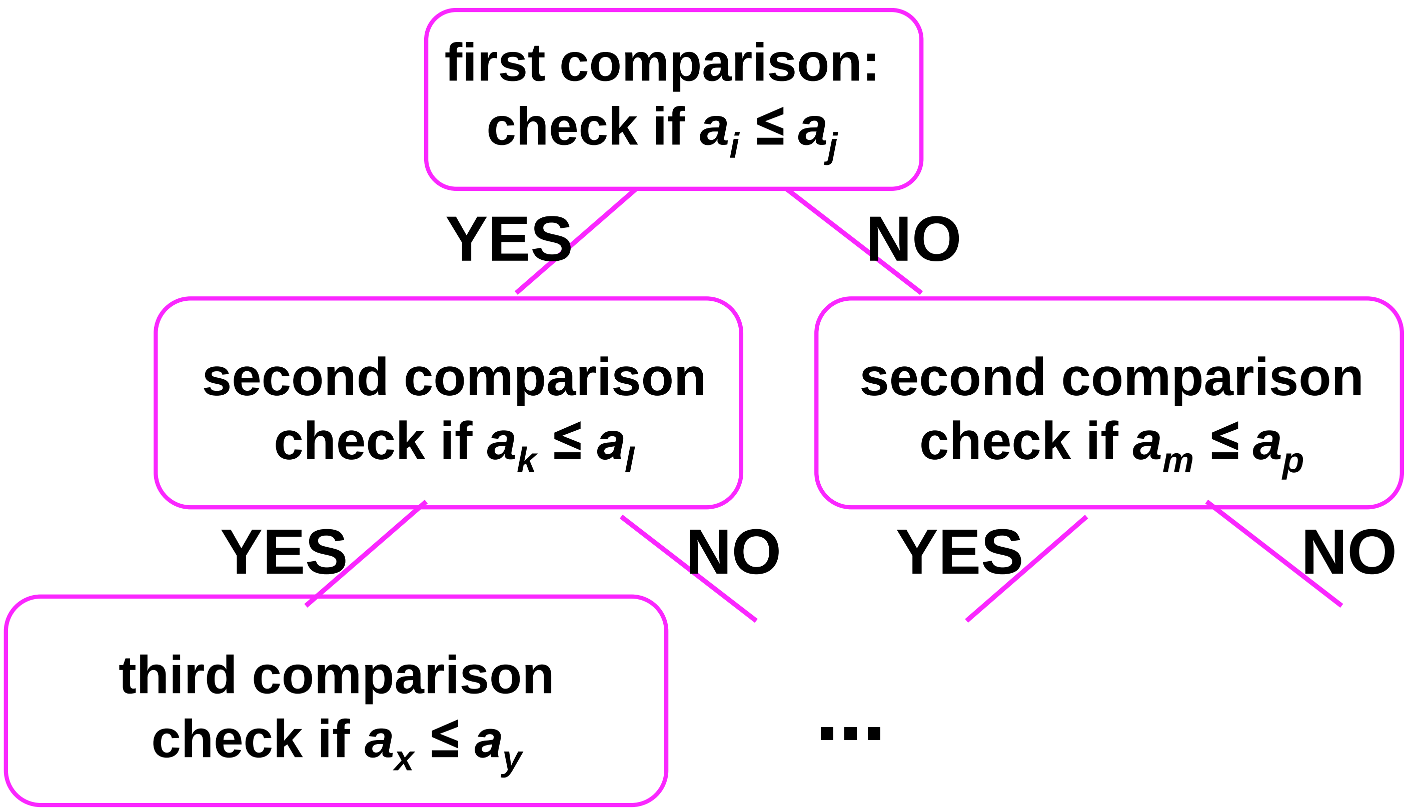
- **Quick sort:**
  - Divide-and-conquer:
    - Partition array into two sub-arrays, recursively sort both
    - All of first sub-array ≤ all of second subarray
    - No merge step needed!
  - Fast in practice
  - $O(n \log n)$ average case
  - $O(n^2)$ worst case (on sorted or reverse-sorted input)
- **Randomized Quicksort:**
  - $O(n^2)$ worst case (on no particular input)
  - $O(n \log n)$ expected running time

HOW FAST CAN WE SORT?

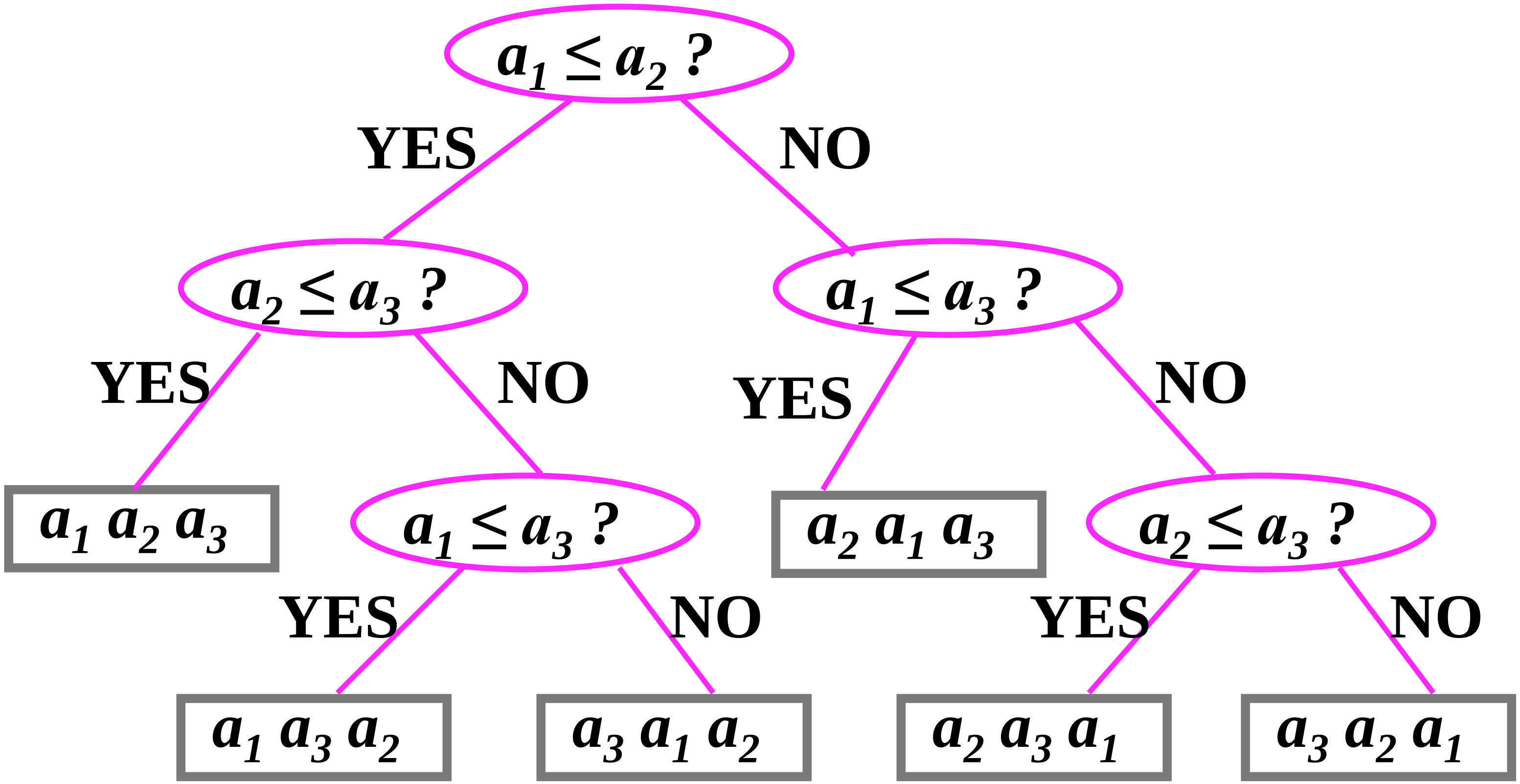
- **We will provide a lower bound, then beat it**
  - ***How do you suppose we can beat impossibility?***
- **Observation: All sorting algorithms so far are *comparison sorts***
  - The only operation used to gain ordering information about a sequence is the **pairwise comparison of two elements**
- **Theorem: All comparison sorts are Ω(n log n)**

DECISION TREE

- A decision tree represents the comparisons made by a comparison sort. Every thing else is ignored.



DECISION TREE FOR INSERTION SORT OF 3 ITEMS



What do the leaves represent?

How many leaves are there? Why?

DECISION TREE

- **Decision trees can model comparison sorts.**
- **For a given algorithm (e.g., Insertion Sort):**
  - One decision tree for each n
  - Tree paths are all possible execution traces
  - ***What's the longest path in a decision tree for insertion sort? For merge sort?***
- ***What is the asymptotic height of any decision tree for sorting n elements?***
  - Answer: $\Omega(n \log n)$ (let's prove it…)

DECISION TREE: HOW MANY LEAVES?

- **Must be at least one leaf for each permutation of the input (Why?)**
  - otherwise there would be a situation that was not correctly sorted
- **Number of permutations of *n* keys is *n!***
- **Decision trees are binary trees.**
- ***Minimum depth of a binary tree with n! leaves?***
- ***Maximum #leaves of a binary tree of height h?***

COMPARISON SORTING LOWER BOUND

Theorem: Any decision tree that sorts n elements has height $\Omega(n \log n)$

Proof: Maximum number of leaves in a binary tree with height h is 2^h.

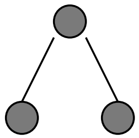
$$2^h \geq n!$$

$$h \geq \log(n!)$$

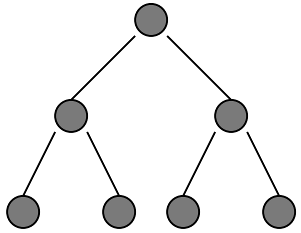
$$= \log(n(n-1)(n-1)\ldots(2)(1))$$

$$\geq (n/2)\log(n/2) \quad \text{(WHY??)}$$

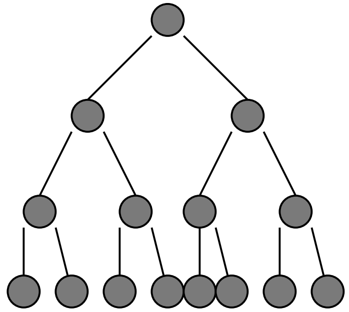
$$= \Omega(n \log n)$$



$h = 1$,
2^1 leaves



$h = 2$, 2^2 leaves



$h = 3$, 2^3 leaves

COMPARISON SORTING LOWER BOUND

- **Time to comparison sort n elements is $\Omega(n \log n)$**
- **Corollary: Heapsort and Mergesort are asymptotically optimal comparison sorts**
  - Quicksort is not asymptotically-optimal. Yet, it is fast in practice.
- **But the name of this lecture is "Linear-Time Sorting"!**
  - ***How can we do better than $\Omega(n \log n)$?***

LINEAR-TIME SORTING: COUNTING SORT

- **No comparisons between elements!**
- ***But…*depends on assumption that the numbers being sorted are in the range *1..k***
  - **where *k* must be *O(n)* for it to take linear time**
- **Input: A[1..*n*], where A[j] $\in$ {1, 2, 3, …, *k*}**
- **Output: sorted array B[1..*n*] (not in-place)**
- **Uses an array C[1..*k*] for auxiliary storage**
  - ***Space Complexity??***

LINEAR-TIME SORTING: COUNTING SORT

```
COUNTINGSORT (A, B, k)
      for i=1 to k
            C[i]= 0;
      for j=1 to n
            C[A[j]] += 1;
      for i=2 to k
            C[i] = C[i] + C[i-1];
      for j=n downto 1
            B[C[A[j]]] = A[j];
            C[A[j]] -= 1;
```

Takes time O(k)

Takes time O(n)

TOTAL TIME: O(n+k)

COUNTING SORT EXAMPLE: LOOP 1

Sort A={4 1 3 4 3} with k = 4

A :

1	2	3	4	5
4	**1**	**3**	**4**	**3**

C :

1	2	3	4
0	**0**	**0**	**0**

B :

for $\boldsymbol{i} \leftarrow \mathbf{1}\ \boldsymbol{to}\ \boldsymbol{k}$

do $\boldsymbol{C}[\ \boldsymbol{i}\ ] \leftarrow \mathbf{0}$

COUNTING SORT EXAMPLE: LOOP 2

A :

1	2	3	4	5
4	1	3	4	3

C :

1	2	3	4
0	0	0	1

B :

for $i \leftarrow 1$ to n

do $C[\,A[\,j\,]\,] \leftarrow C[\,A[\,j\,]] + 1$ $\quad \triangleright \quad C[\,i\,] = |\{\,key \le i\,\}|$

COUNTING SORT EXAMPLE: LOOP 2

A :

1	2	3	4	5
4	1	3	4	3

C :

1	2	3	4
1	0	0	1

B :

for $\boldsymbol{i} \leftarrow \mathbf{1}$ $\boldsymbol{to}$ $\boldsymbol{n}$

do $\boldsymbol{C}[\,\boldsymbol{A}[\,\boldsymbol{j}\,]\,] \leftarrow \boldsymbol{C}[\,\boldsymbol{A}[\,\boldsymbol{j}\,]] + \mathbf{1}$ $\quad \triangleright \quad \boldsymbol{C}[\,\boldsymbol{i}\,] = |\{\,\boldsymbol{key} \le \boldsymbol{i}\,\}|$

COUNTING SORT EXAMPLE: LOOP 2

A :

1	2	3	4	5
4	1	3	4	3

C :

1	2	3	4
1	0	1	1

B :

for $i \leftarrow 1\ to\ n$

  do $C[\,A[\,j\,]\,] \leftarrow C[\,A[\,j\,]] + 1$ $\qquad \triangleright \quad C[\,i\,] = |\{\,key \le i\,\}|$

COUNTING SORT EXAMPLE: LOOP 2

A :

1	2	3	4	5
4	1	3	4	3

C :

1	2	3	4
1	0	1	2

B :

for $i \leftarrow 1$ to n

do $C[\,A[\,j\,]\,] \leftarrow C[\,A[\,j\,]] + 1$ $\quad \triangleright \quad C[\,i\,] = |\{\,key \le i\,\}|$

COUNTING SORT EXAMPLE: LOOP 2

A :

1	2	3	4	5
4	1	3	4	3

C :

1	2	3	4
1	0	2	2

B :

for $\boldsymbol{i} \leftarrow \mathbf{1}\ \boldsymbol{to}\ \boldsymbol{n}$

do $\boldsymbol{C}[\,\boldsymbol{A}[\,\boldsymbol{j}\,]\,] \leftarrow \boldsymbol{C}[\,\boldsymbol{A}[\,\boldsymbol{j}\,]] + \mathbf{1}$ $\quad \triangleright \quad \boldsymbol{C}[\,\boldsymbol{i}\,] = |\{\,\boldsymbol{key} \le \boldsymbol{i}\,\}|$

COUNTING SORT EXAMPLE: LOOP 3

A :

1	2	3	4	5
4	**1**	**3**	**4**	**3**

C :

1	2	3	4
1	**0**	**2**	**2**

B :

for $i \leftarrow 2$ to k
  do $C[\,i\,] \leftarrow C[\,i\,] + C[\,i - 1]$ ▷ $C[\,i\,] = |\{\,key \le i\,\}|$

COUNTING SORT EXAMPLE: LOOP 3

A :

1	2	3	4	5
4	**1**	**3**	**4**	**3**

C :

1	2	3	4
1	**1**	**2**	**2**

B :

for $i \leftarrow 2$ to k

do $C[\,i\,] \leftarrow C[\,i\,] + C[\,i - 1]$ ▷ $C[\,i\,] = |\{\, key \le i\,\}|$

COUNTING SORT EXAMPLE: LOOP 3

A:

1	2	3	4	5
4	1	3	4	3

C:

1	2	3	4
1	1	3	2

B:

for $i \leftarrow 2$ *to* k

  do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{key \leq i\}|$

COUNTING SORT EXAMPLE: LOOP 3

A :

1	2	3	4	5
4	1	3	4	3

C :

1	2	3	4
1	1	3	5

B :

for $i \leftarrow 2$ to k

do $C[\,i\,] \leftarrow C[\,i\,] + C[\,i - 1]$ $\triangleright$ $C[\,i\,] = |\{\,key \le i\,\}|$

COUNTING SORT EXAMPLE: LOOP 3

A :

1	2	3	4	5
4	1	3	4	3

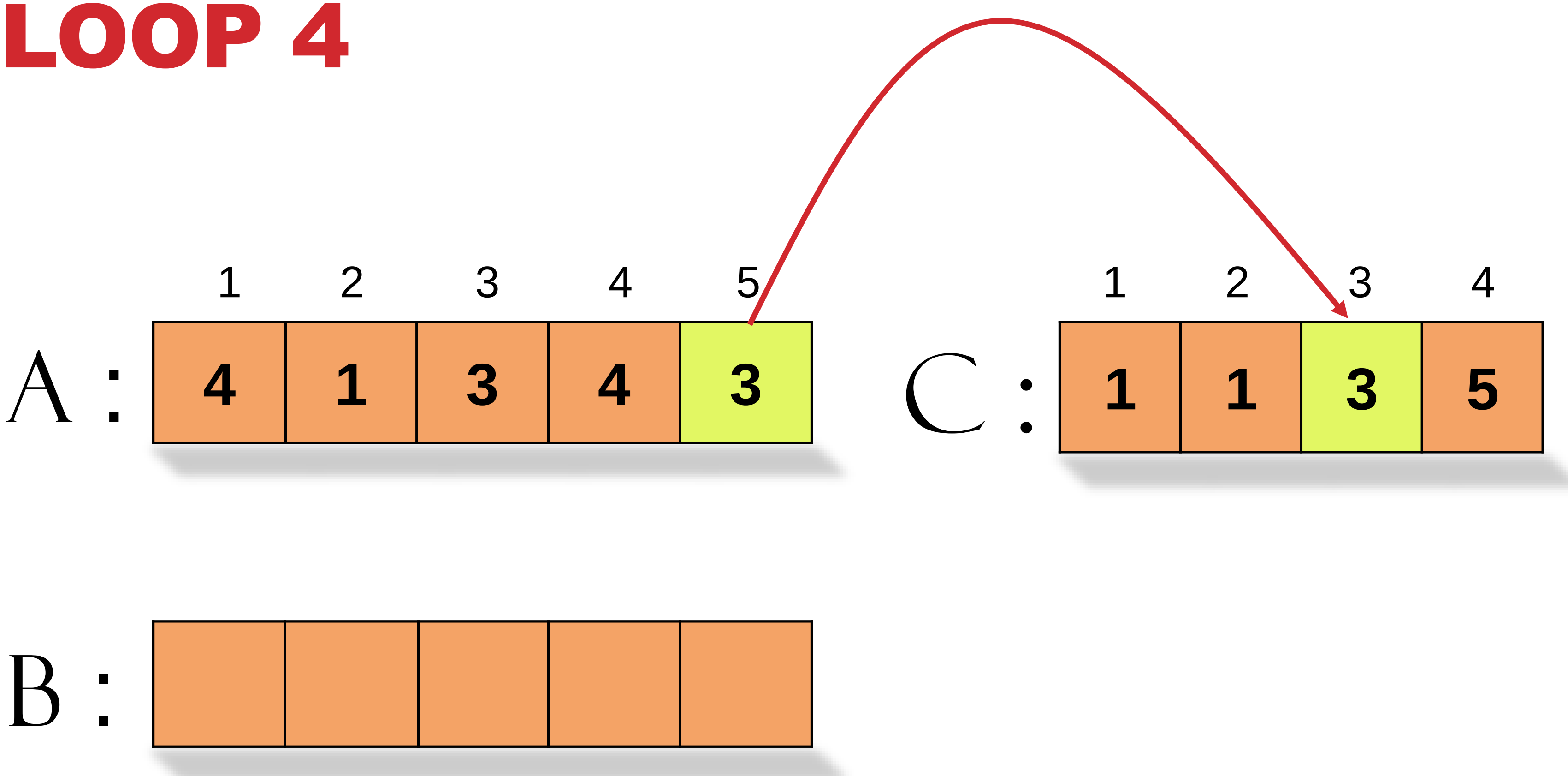
C :

1	2	3	4
1	1	3	5

B :

for $\boldsymbol{i \leftarrow 2\ to\ k}$
do $\boldsymbol{C[i] \leftarrow C[i] + C[i-1]}$ ▷ $\boldsymbol{C[i] = |\{key \leq i\}|}$

COUNTING SORT EXAMPLE: LOOP 4

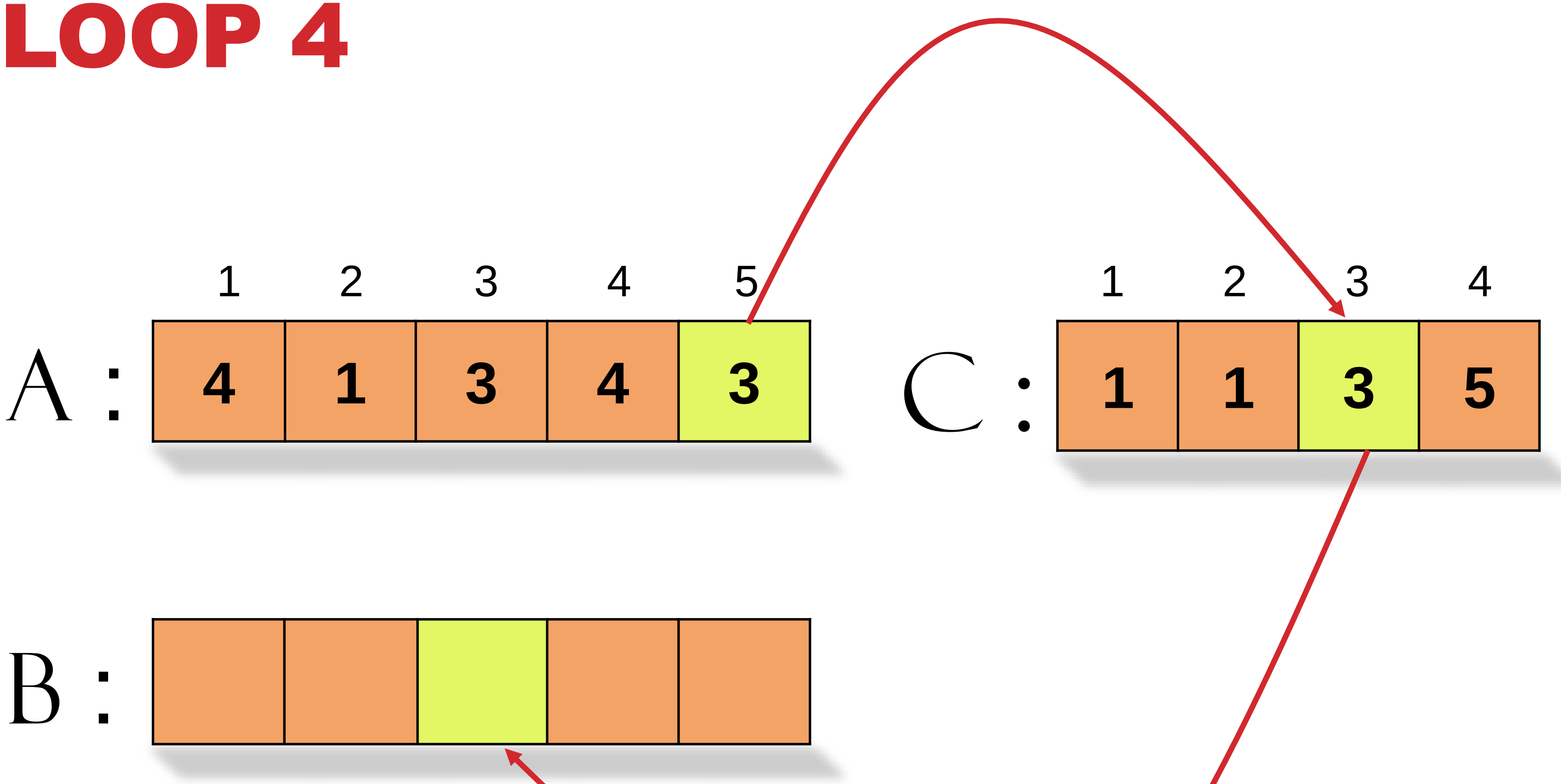


$\textbf{for } j \leftarrow n \textit{ downto } 1$

$\quad B[\,C[\,A[\,j\,]\,]\,] \leftarrow A[\,j\,]$

$\quad C[\,A[\,j\,]\,] \leftarrow C[\,A[\,j\,]\,] - 1$

COUNTING SORT EXAMPLE: LOOP 4

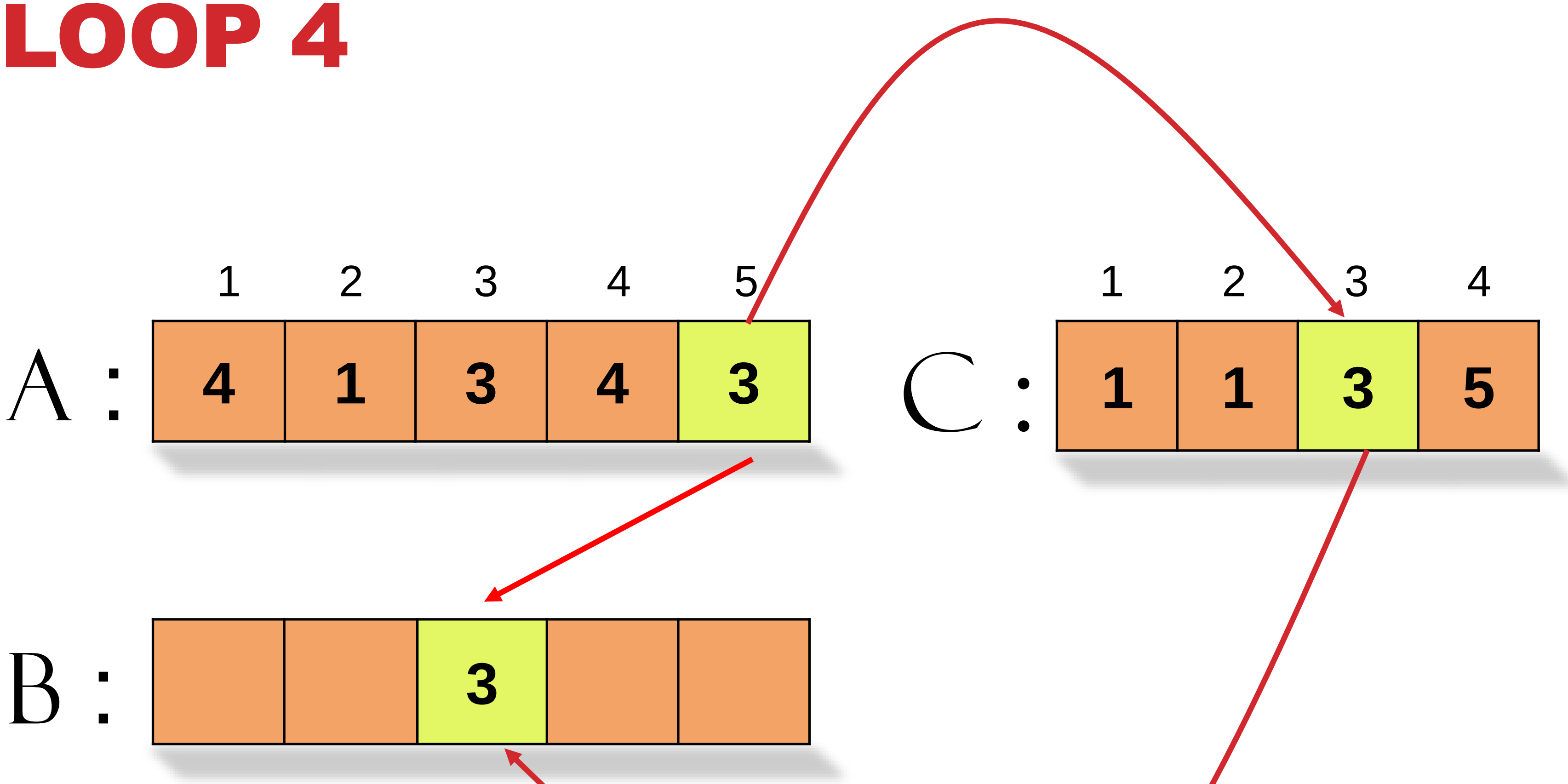


for $j \leftarrow n$ *downto* 1

$\quad B[\,C[\,A[\,j\,]\,]\,] \leftarrow A[\,j\,]$

$\quad C[\,A[\,j\,]\,] \leftarrow C[\,A[\,j\,]\,] - 1$

COUNTING SORT EXAMPLE: LOOP 4



for $j \leftarrow n$ *downto* 1

$B[\,C[\,A[\,j\,]\,]\,] \leftarrow A[\,j\,]$

$C[\,A[\,j\,]\,] \leftarrow C[\,A[\,j\,]\,] - 1$

COUNTING SORT EXAMPLE: LOOP 4

A :

1	2	3	4	5
4	1	3	4	3

C :

1	2	3	4
1	1	2	5

B :

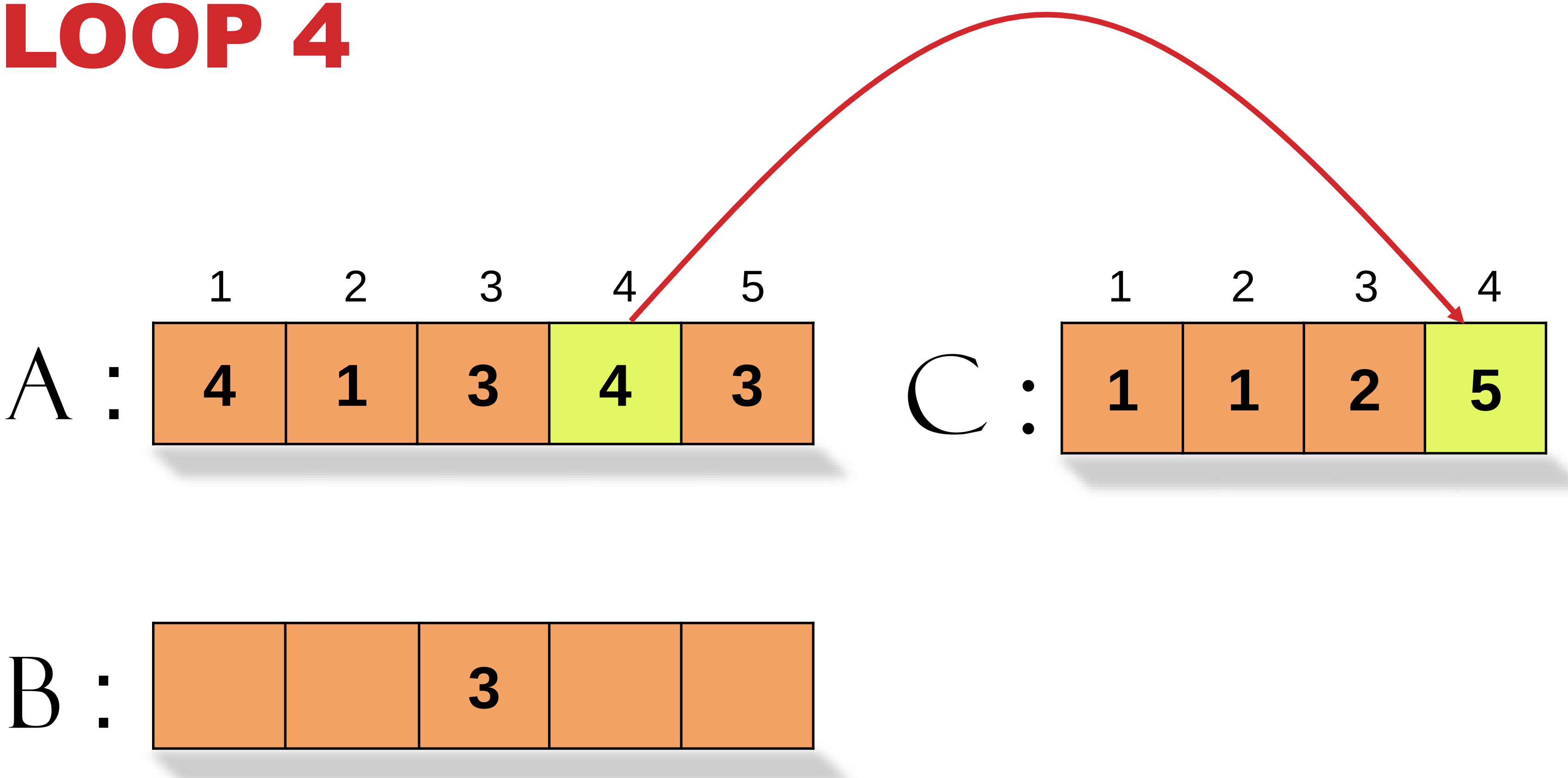
		3		

for $j \leftarrow n$ *downto* 1

$B[\,C[\,A[\,j\,]\,]\,] \leftarrow A[\,j\,]$

$C[\,A[\,j\,]\,] \leftarrow C[\,A[\,j\,]\,] - 1$

COUNTING SORT EXAMPLE: LOOP 4

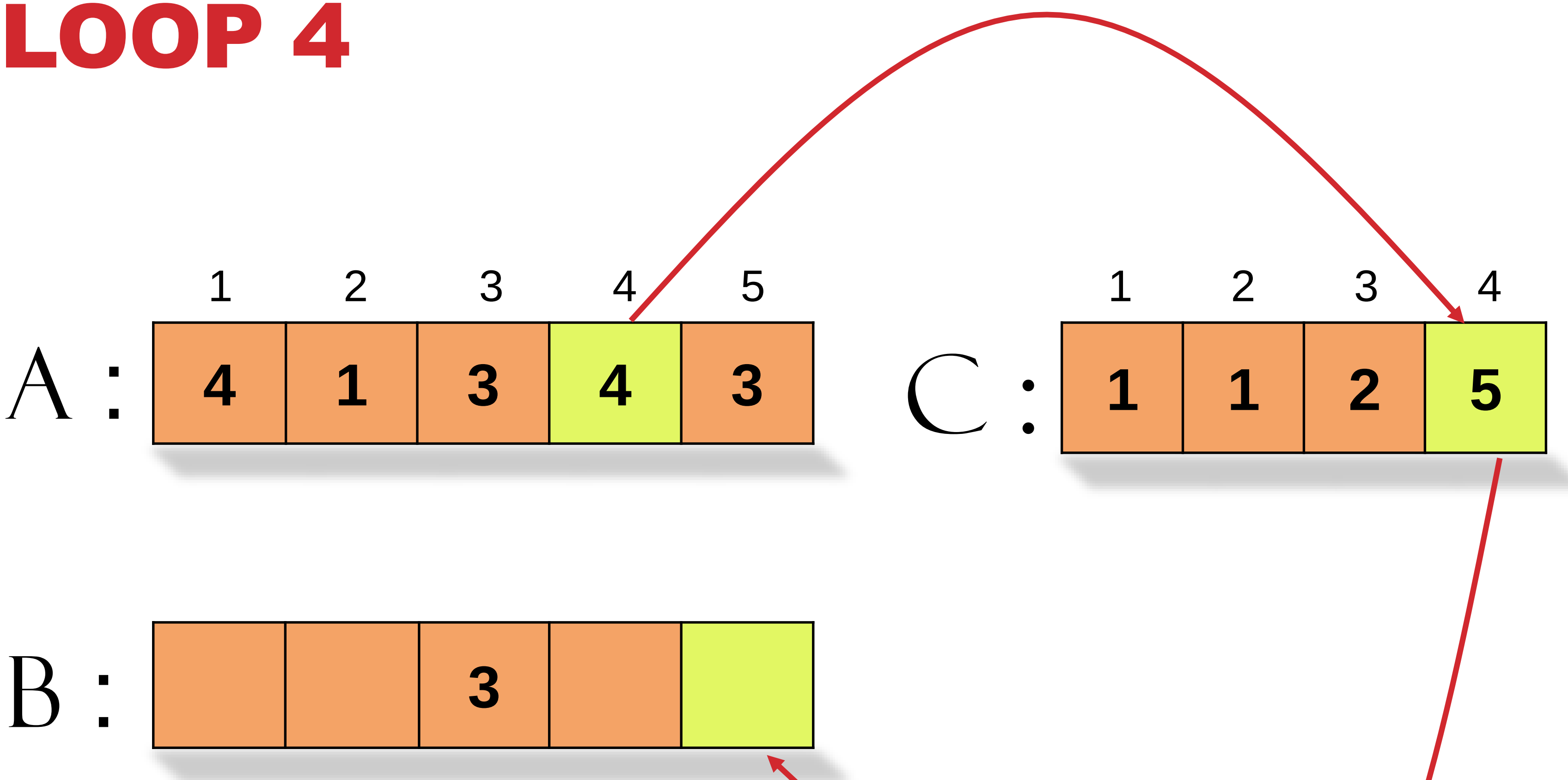


for $j \leftarrow n$ ***downto*** $\mathbf{1}$

$$\mathbf{B}[\,\boldsymbol{C}[\,\boldsymbol{A}[\,\boldsymbol{j}\,]\,]\,] \leftarrow \boldsymbol{A}[\,\boldsymbol{j}\,]$$

$$\boldsymbol{C}[\,\boldsymbol{A}[\,\boldsymbol{j}\,]\,] \leftarrow \boldsymbol{C}[\,\boldsymbol{A}[\,\boldsymbol{j}\,]\,] - \mathbf{1}$$

COUNTING SORT EXAMPLE: LOOP 4



for $j \leftarrow n$ *downto* 1

$$B[\,C[\,A[\,j\,]\,]\,] \leftarrow A[\,j\,]$$
$$C[\,A[\,j\,]\,] \leftarrow C[\,A[\,j\,]\,] - 1$$

COUNTING SORT EXAMPLE: LOOP 4

A :

1	2	3	4	5
4	1	3	4	3

C :

1	2	3	4
1	1	2	5

B :

		3		4

for $j \leftarrow n$ *downto* 1

$$B[\,C[\,A[\,j\,]\,]\,] \leftarrow A[\,j\,]$$

$$C[\,A[\,j\,]\,] \leftarrow C[\,A[\,j\,]\,] - 1$$

COUNTING SORT EXAMPLE: LOOP 4

A :

1	2	3	4	5
4	1	3	4	3

C :

1	2	3	4
1	1	2	4

B :

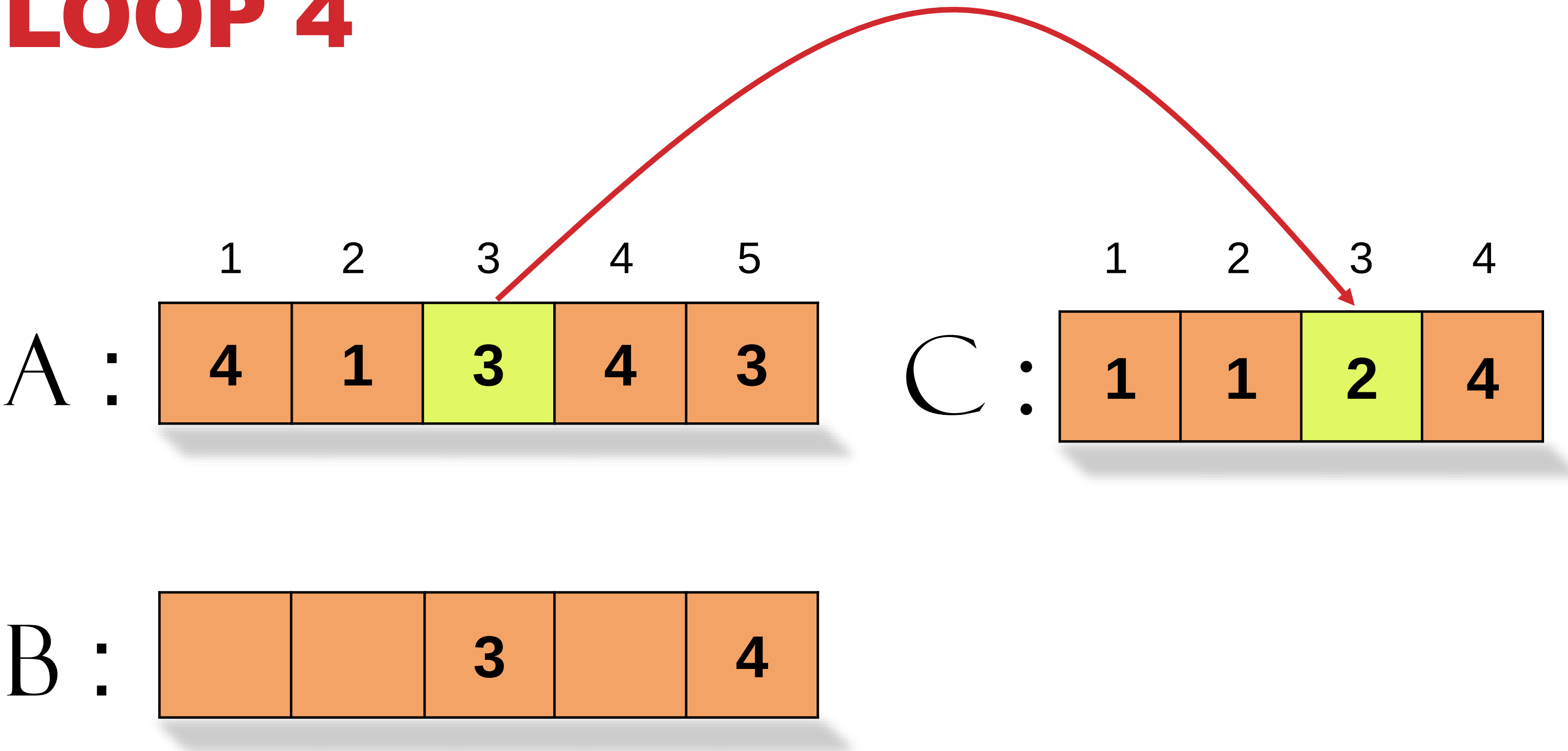
		3		4

for $j \leftarrow n$ *downto* 1

$$B[\,C[\,A[\,j\,]\,]\,] \leftarrow A[\,j\,]$$

$$C[\,A[\,j\,]\,] \leftarrow C[\,A[\,j\,]\,] - 1$$

COUNTING SORT EXAMPLE: LOOP 4

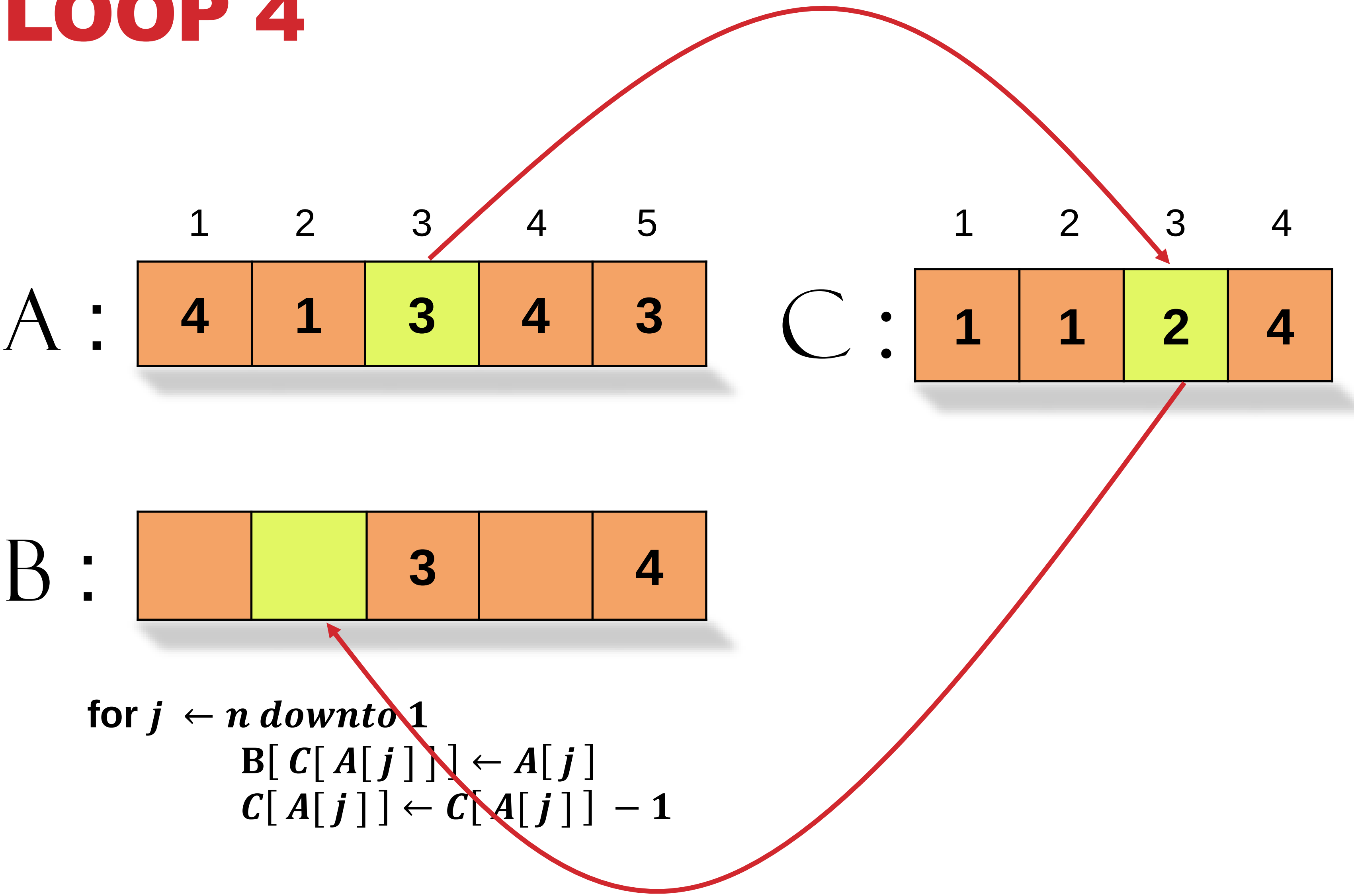


for $j \leftarrow n$ ***downto*** **1**

$\mathbf{B}[\,\boldsymbol{C}[\,\boldsymbol{A}[\,\boldsymbol{j}\,]\,]\,] \leftarrow \boldsymbol{A}[\,\boldsymbol{j}\,]$

$\boldsymbol{C}[\,\boldsymbol{A}[\,\boldsymbol{j}\,]\,] \leftarrow \boldsymbol{C}[\,\boldsymbol{A}[\,\boldsymbol{j}\,]\,] - \mathbf{1}$

COUNTING SORT EXAMPLE: LOOP 4

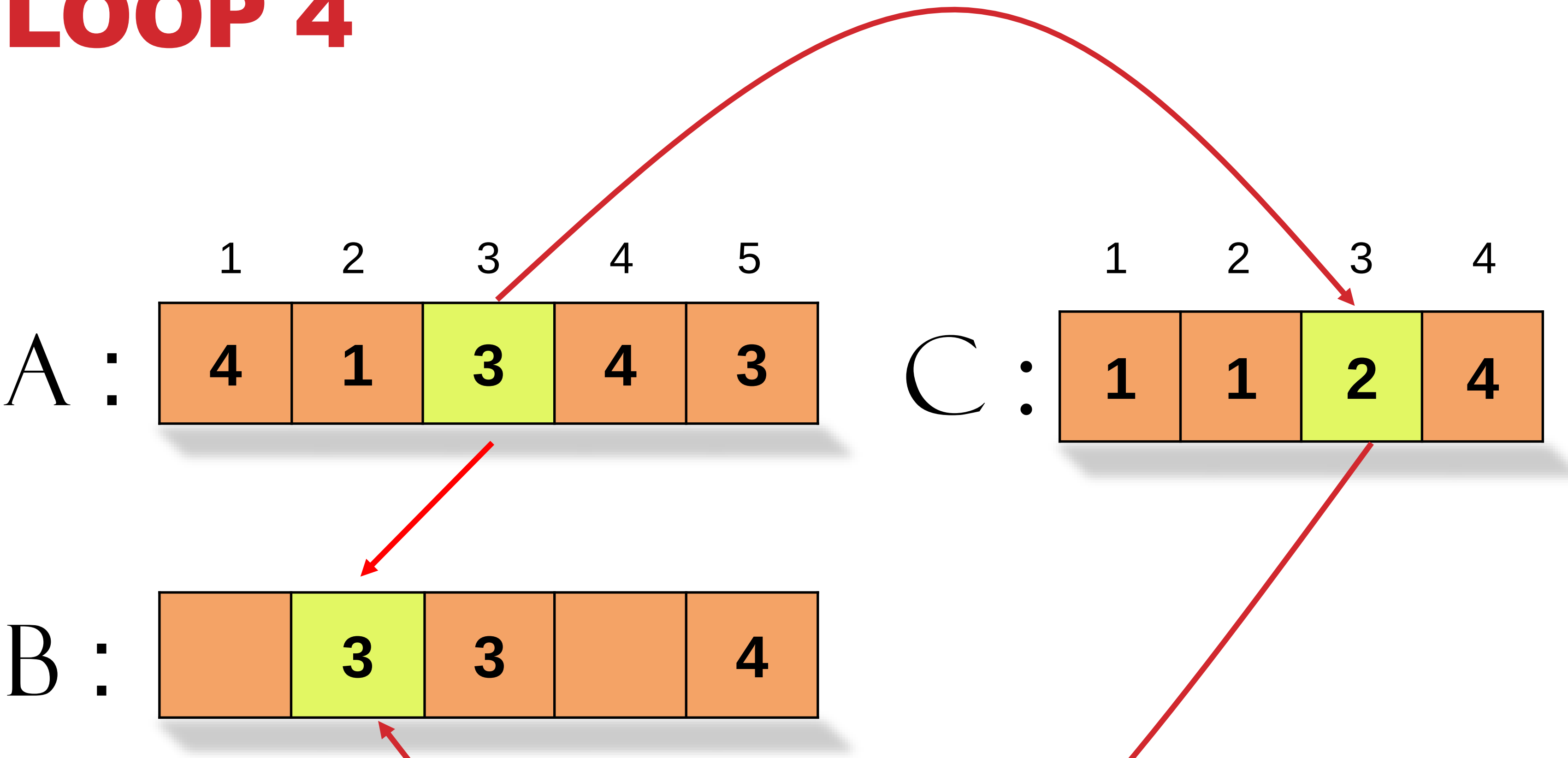


for $j \leftarrow n\ downto\ 1$

$B[\,C[\,A[\,j\,]\,]\,] \leftarrow A[\,j\,]$

$C[\,A[\,j\,]\,] \leftarrow C[\,A[\,j\,]\,] - 1$

COUNTING SORT EXAMPLE: LOOP 4



for $j \leftarrow n$ ***downto*** **1**

$\mathbf{B}[\,\boldsymbol{C}[\,\boldsymbol{A}[\,\boldsymbol{j}\,]\,]\,] \leftarrow \boldsymbol{A}[\,\boldsymbol{j}\,]$

$\boldsymbol{C}[\,\boldsymbol{A}[\,\boldsymbol{j}\,]\,] \leftarrow \boldsymbol{C}[\,\boldsymbol{A}[\,\boldsymbol{j}\,]\,] - \mathbf{1}$

COUNTING SORT EXAMPLE: LOOP 4

A :

1	2	3	4	5
4	1	3	4	3

C :

1	2	3	4
1	1	1	4

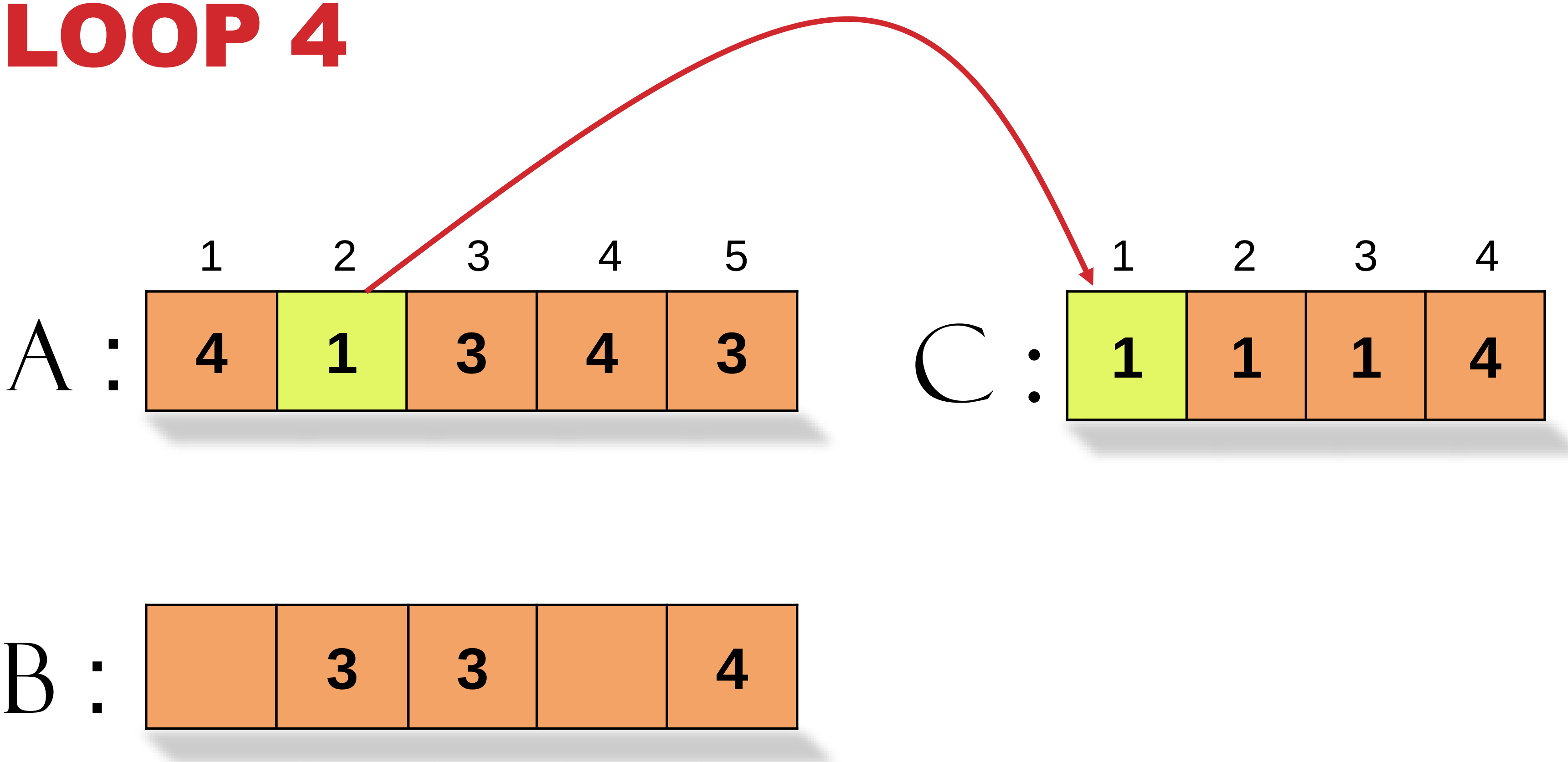
B :

	3	3		4

for $j \leftarrow n$ $downto$ 1

$$B[\,C[\,A[\,j\,]\,]\,] \leftarrow A[\,j\,]$$
$$C[\,A[\,j\,]\,] \leftarrow C[\,A[\,j\,]\,] - 1$$

COUNTING SORT EXAMPLE: LOOP 4



for $j \leftarrow n$ $\boldsymbol{downto}$ $\mathbf{1}$

$$\mathbf{B}[\,\boldsymbol{C}[\,\boldsymbol{A}[\,\boldsymbol{j}\,]\,]\,] \leftarrow \boldsymbol{A}[\,\boldsymbol{j}\,]$$

$$\boldsymbol{C}[\,\boldsymbol{A}[\,\boldsymbol{j}\,]\,] \leftarrow \boldsymbol{C}[\,\boldsymbol{A}[\,\boldsymbol{j}\,]\,] - \mathbf{1}$$

COUNTING SORT EXAMPLE: LOOP 4

A :

1	2	3	4	5
4	1	3	4	3

C :

1	2	3	4
1	1	1	4

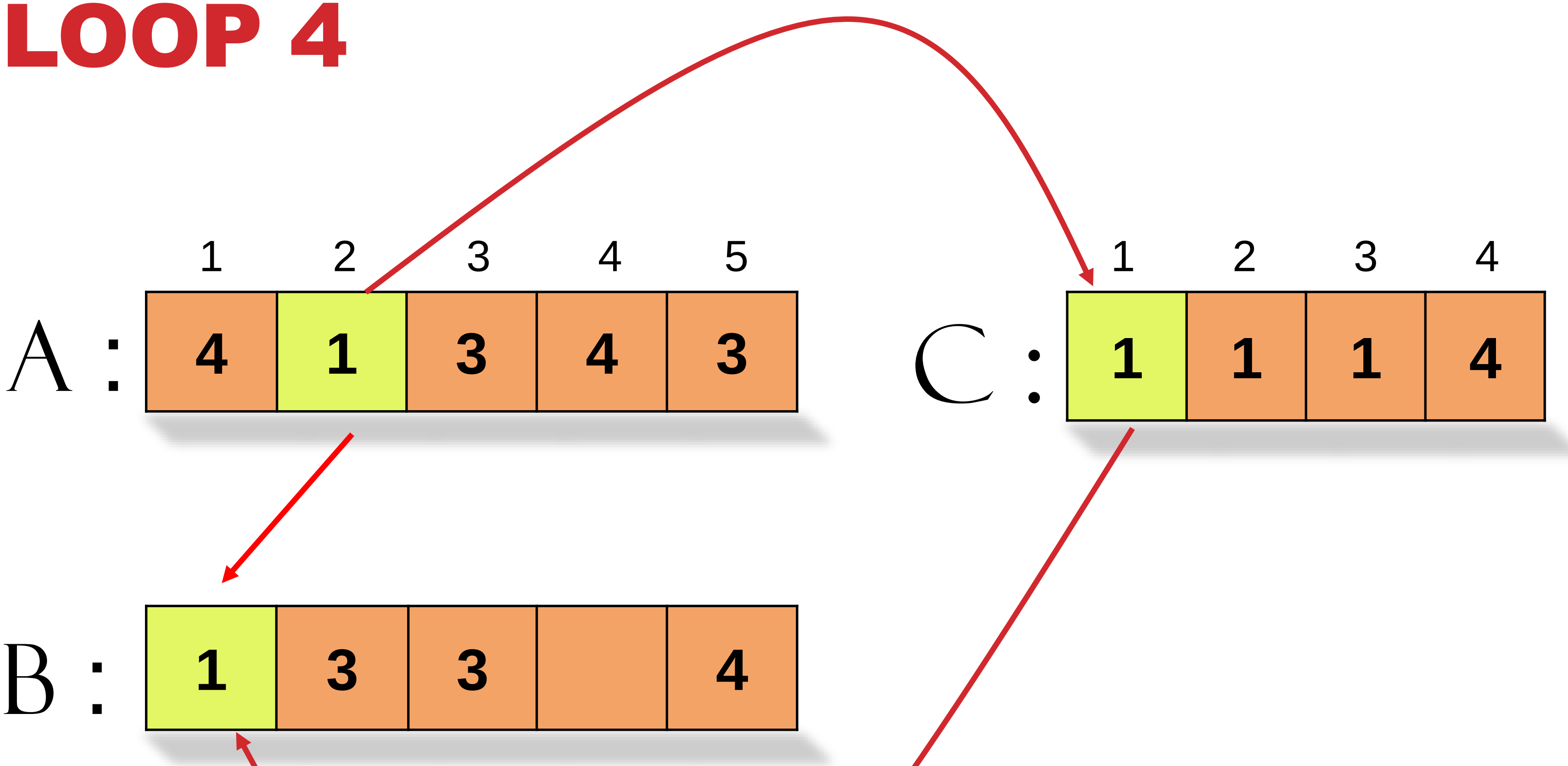
B :

	3	3		4

for $j \leftarrow n$ *downto* 1

$$B[\,C[\,A[\,j\,]\,]\,] \leftarrow A[\,j\,]$$
$$C[\,A[\,j\,]\,] \leftarrow C[\,A[\,j\,]\,] - 1$$

COUNTING SORT EXAMPLE: LOOP 4



for $j \leftarrow n$ *downto* 1

$\quad B[\,C[\,A[\,j\,]\,]\,] \leftarrow A[\,j\,]$

$\quad C[\,A[\,j\,]\,] \leftarrow C[\,A[\,j\,]\,] - 1$

COUNTING SORT EXAMPLE: LOOP 4

A :

1	2	3	4	5
4	1	3	4	3

C :

1	2	3	4
0	1	1	4

B :

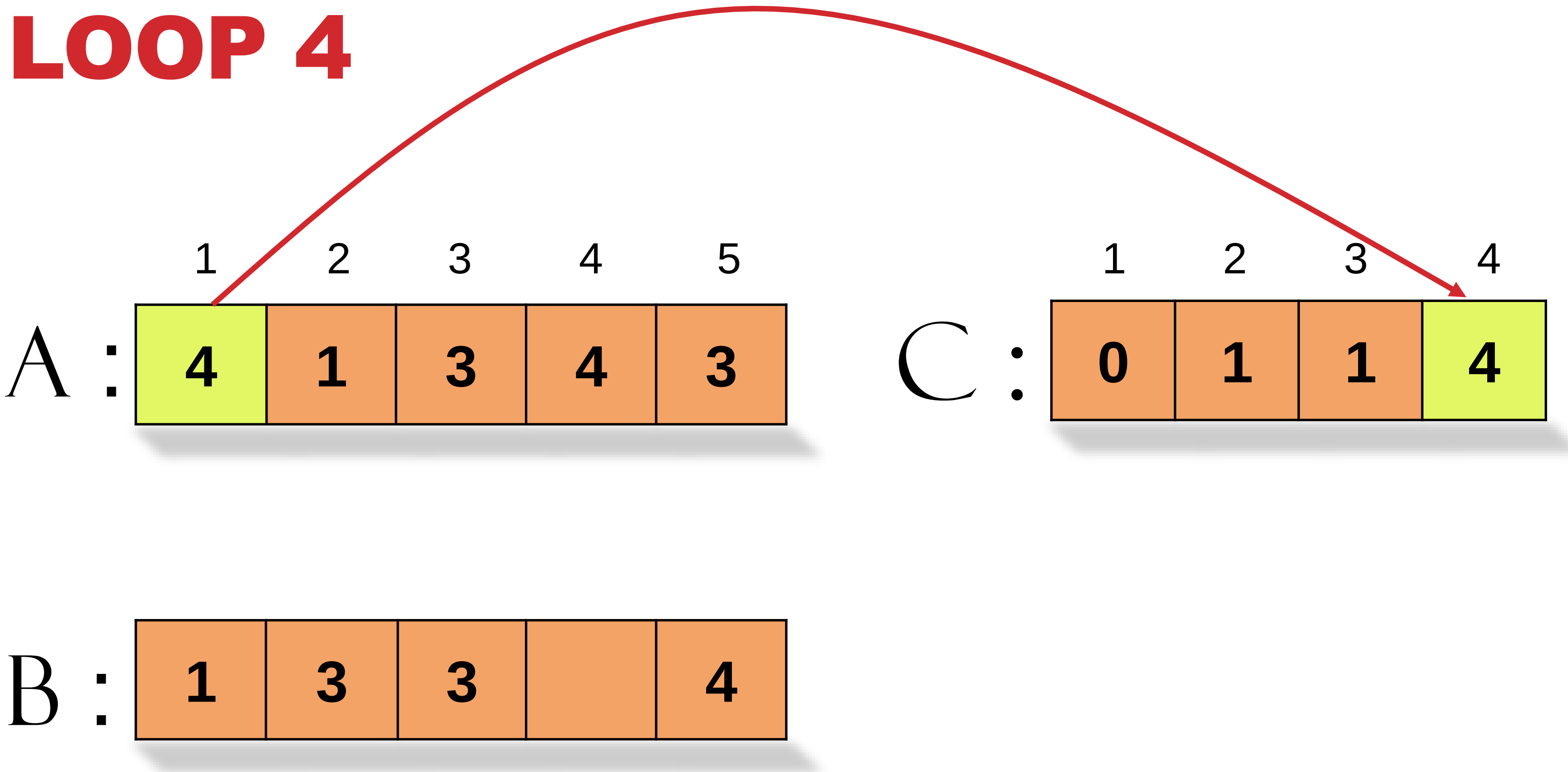
1	3	3		4

for $j \leftarrow n$ *downto* 1

$$\mathbf{B}[\,C[\,A[\,j\,]\,]\,] \leftarrow A[\,j\,]$$

$$C[\,A[\,j\,]\,] \leftarrow C[\,A[\,j\,]\,] - 1$$

COUNTING SORT EXAMPLE: LOOP 4



for $j \leftarrow n$ ***downto*** **1**

$\mathbf{B}[\,C[\,A[\,j\,]\,]\,] \leftarrow A[\,j\,]$

$C[\,A[\,j\,]\,] \leftarrow C[\,A[\,j\,]\,] - 1$

COUNTING SORT EXAMPLE: LOOP 4

A :

1	2	3	4	5
4	1	3	4	3

C :

1	2	3	4
0	1	1	4

B :

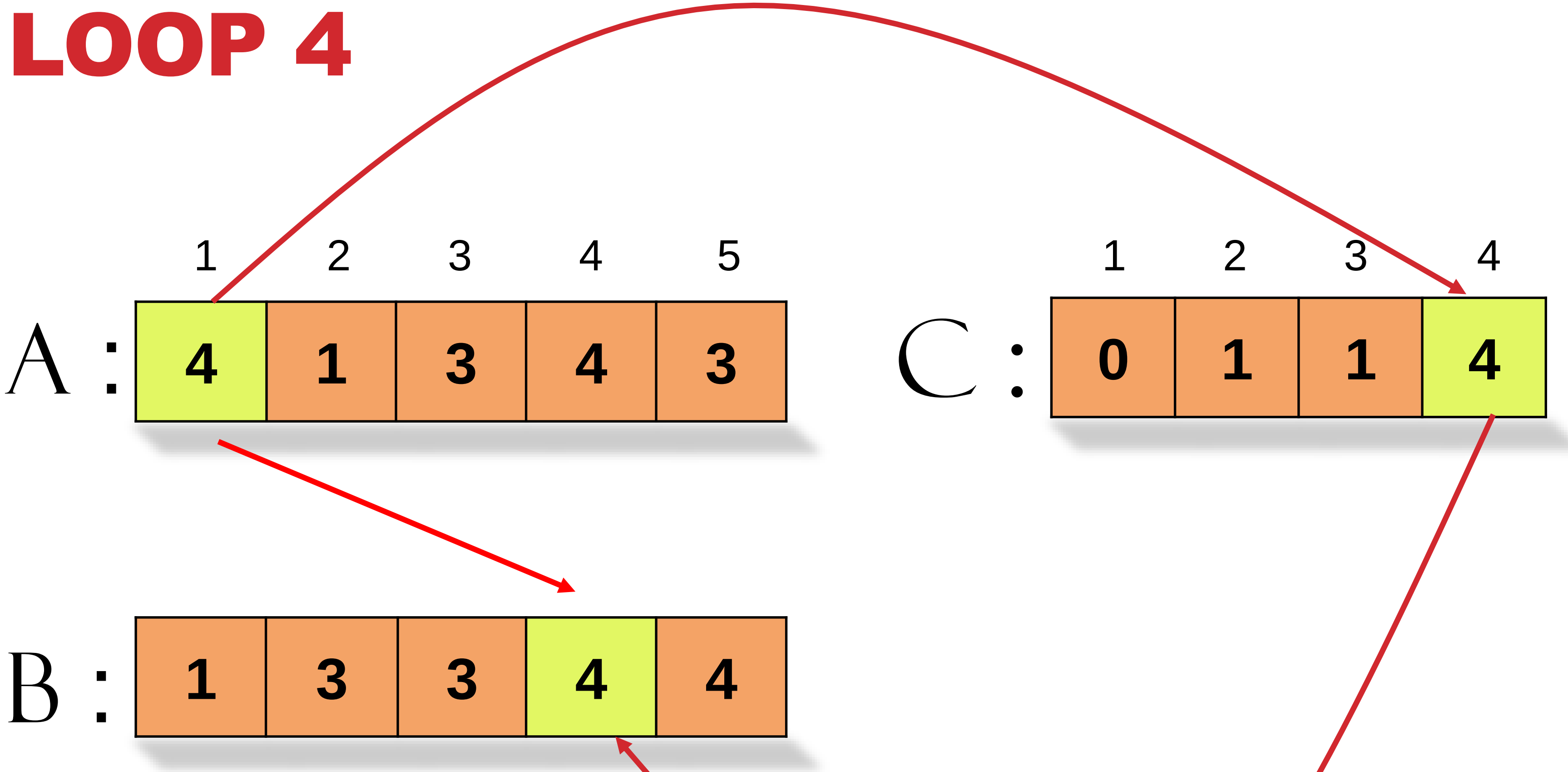
1	3	3		4

for $j \leftarrow n \; downto \; 1$

$\mathbf{B}[\, C[\, A[\, j\,]\,]\,] \leftarrow A[\, j\,]$

$C[\, A[\, j\,]\,] \leftarrow C[\, A[\, j\,]\,] - 1$

COUNTING SORT EXAMPLE: LOOP 4



for $j \leftarrow n\ \mathbf{downto}\ 1$

$\quad \mathbf{B}[\,C[\,A[\,j\,]\,]\,] \leftarrow A[\,j\,]$

$\quad C[\,A[\,j\,]\,] \leftarrow C[\,A[\,j\,]\,] - 1$

COUNTING SORT EXAMPLE: LOOP 4

A :

1	2	3	4	5
4	1	3	4	3

C :

1	2	3	4
0	1	1	3

B :

1	3	3	4	4

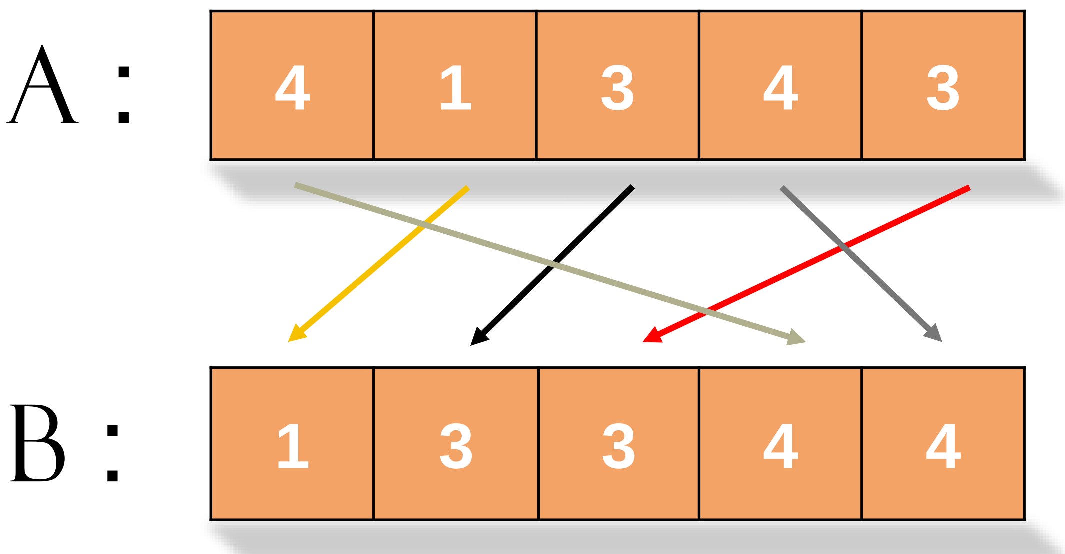
for $j \leftarrow n$ ***downto*** 1

$$\mathbf{B}[\,C[\,A[\,j\,]\,]\,] \leftarrow A[\,j\,]$$

$$C[\,A[\,j\,]\,] \leftarrow C[\,A[\,j\,]\,] - 1$$

STABLE SORTING

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



What other sorts have this property?

COUNTING SORT

- **Cool!** ***Why don't we always use counting sort?***
  - Because it depends on range *k* of elements
- ***Can we use counting sort to sort 32-bit integers? Why or why not?***
  - Answer: NO, *k* is too large (2^{32} = 4,294,967,296)
  - Affects both time and space complexity

RADIX SORT

- Sorting *d-digit* numbers:
  - Sort on the *most significant digit*
  - Then sort on the *second-most significant digit*, etc.
- **Problem:** lots of intermediate results to keep track of
- **Key idea of IBM:** sort the *least significant digit* first
  - Sort a *d-digit* number:

```
RADIXSORT(A, d)
    for i=1 to d
        STABLESORT(A) on digit i
```

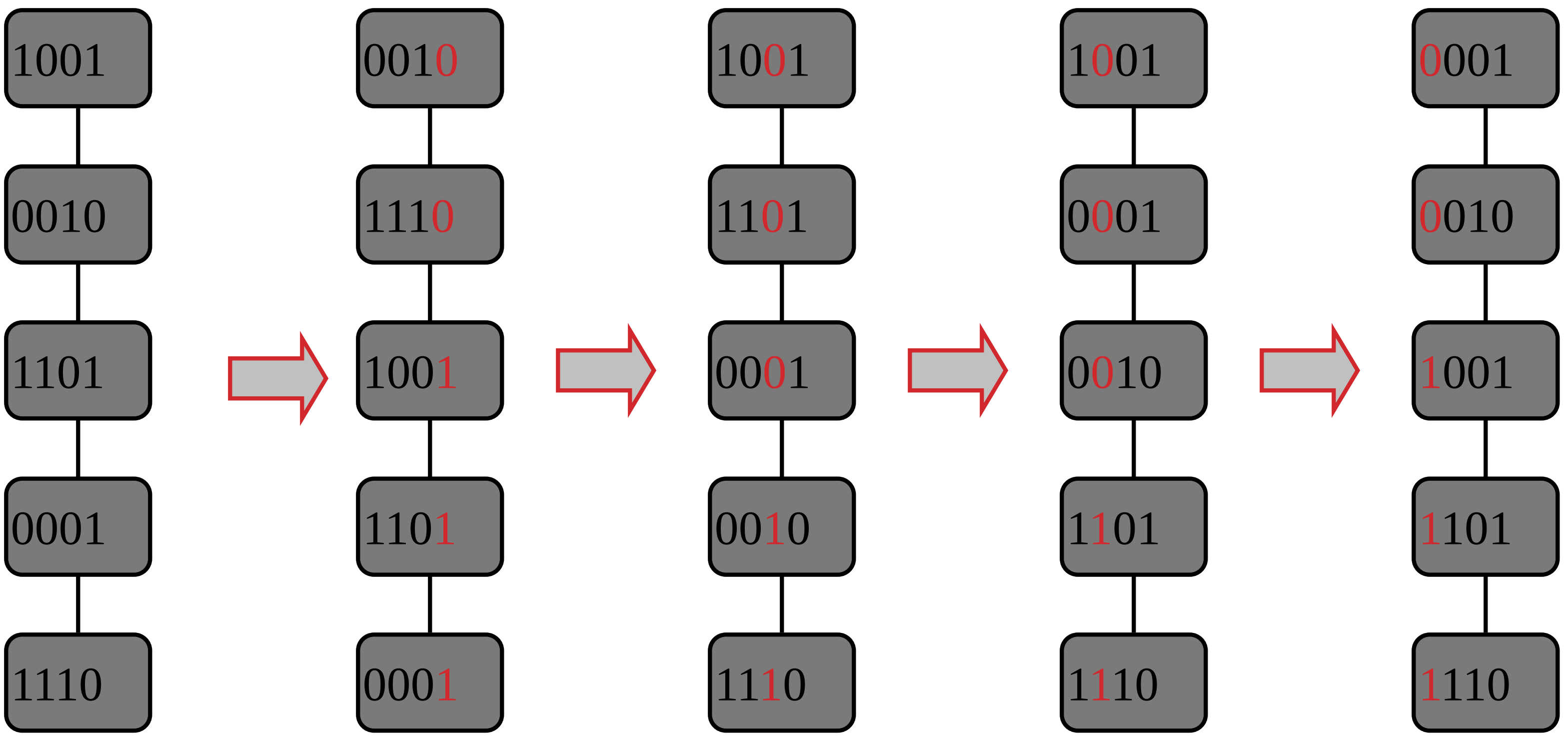
- ***What sort will we use to sort on digits?***

RADIX SORT EXAMPLE: DECIMAL

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

RADIX SORT EXAMPLE: BINARY

- **Sorting a sequence of 4-bit integers**



http://www.cs.usfca.edu/~galles/visualization/RadixSort.html

RADIX SORT

- **Each of the *d* passes over *n* numbers takes time $O(n+k)$, so total time $O(\, d(n+k) \,)$**
  - When *d* is constant (e.g., d = 32 for 32-bit integers) and $k=O(n)$, takes total $O(n)$ time
- **In practice**
  - Radix Sort is fast for large inputs.
  - Radix Sort is simple to code and maintain.
  - Problem: Radix Sort displays little locality of reference (same problem as Heap Sort)
  - A well-tuned quicksort is better, since it runs mostly on consecutive memory locations.

CONCLUSIONS

- **All theorems rely on assumptions to be true:**
  - Example: Sorting is $\Omega(n \log n)$
    - Assumes comparison-based sorting
- **When you come up with an impossibiliy result, try to think outside of the box and find a completely different approach.**
  - Example: Assume different distribution on input, or impose a different condition
    - Counting Sort assumes all items are less than $k = O(n)$
    - Randomized Quicksort makes sure average-case is like best-case
- **Quicksort is quick!**
  - Asymptotic complexity matters, but algorithmic details and computer architecture also matters!