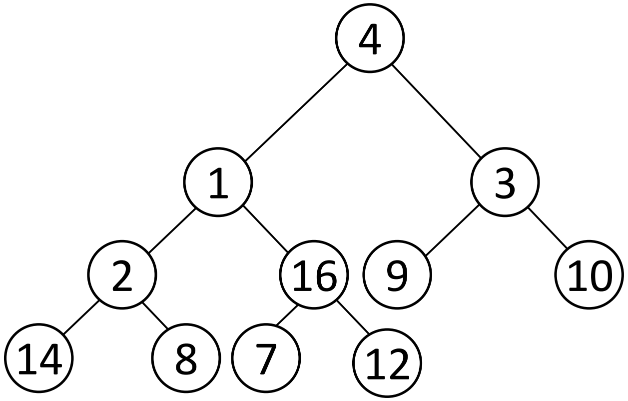
CS-204

Sorting

Heap Sort

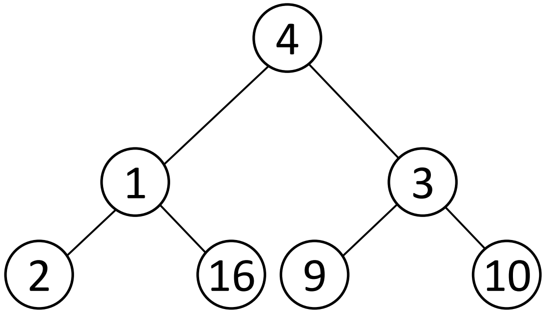
Special Types of Trees

- *Def:* **Full binary tree** = a binary tree in which each node is either a leaf or has degree exactly 2.



Full binary tree

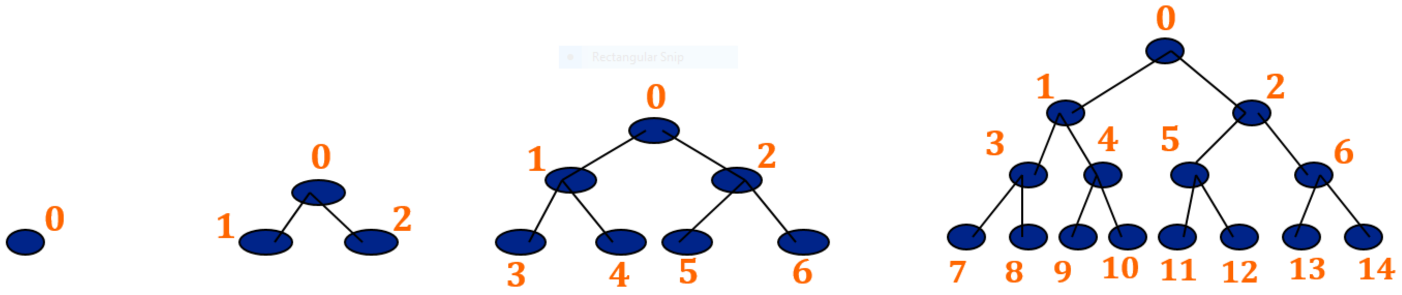
- *Def:* **Complete binary tree** = a binary tree in which all leaves are on the same level and all internal nodes have degree 2.



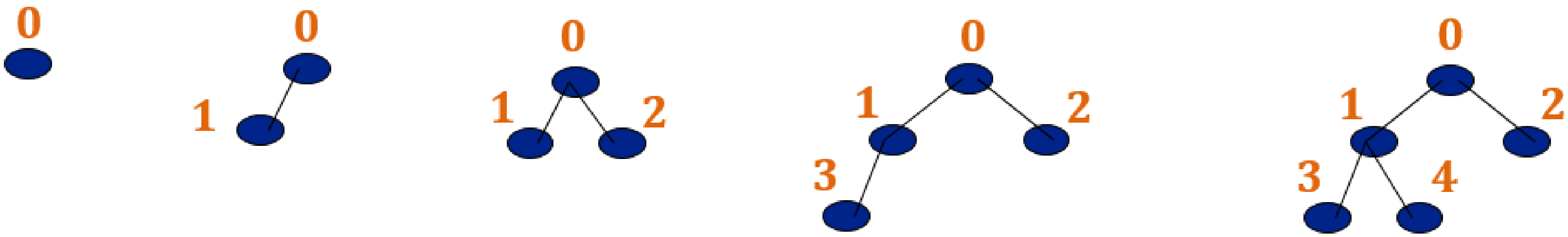
Complete binary tree

Almost Complete Binary Tree

- Canonical labeling of nodes: Label the Nodes in the level-wise fashion from left to right, as shown below

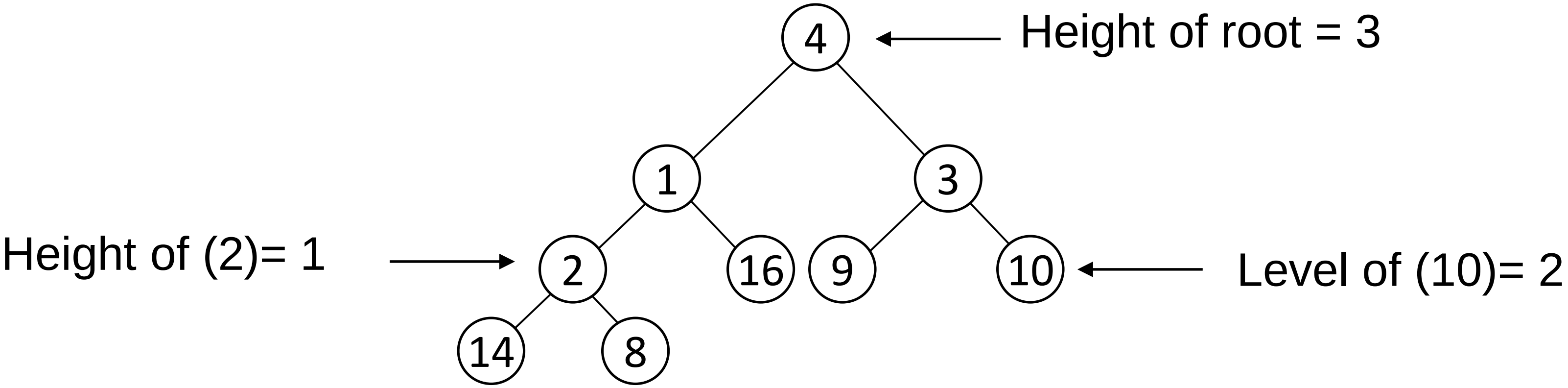


- **Almost Complete Binary Tree**: A binary tree made up of the first n nodes of a canonically labeled complete Binary Tree is called Almost Complete Binary Tree.



Definitions

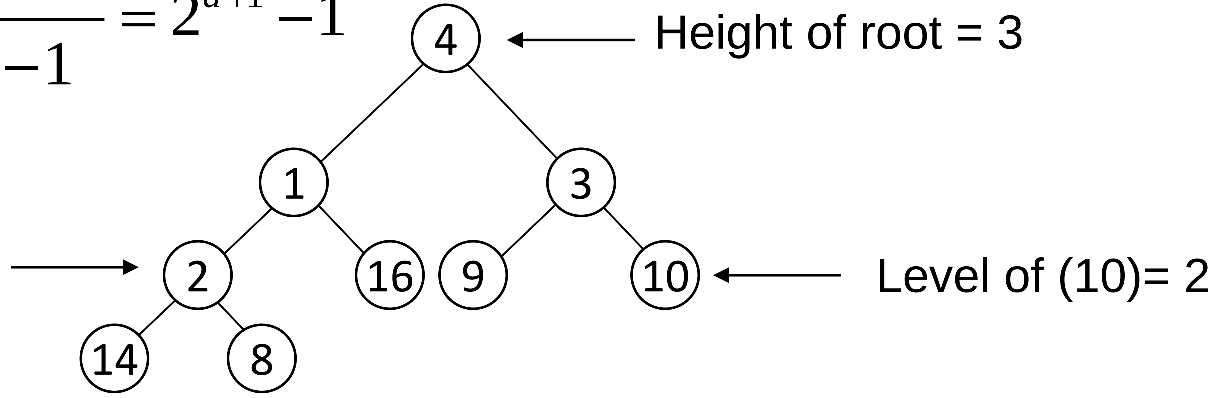
- **Height** of a node = the **number of edges** on the **longest** simple path from the node down to a leaf
- **Level** of a node = the length of a path from the root to the node
- **Height** of tree = height of root node



Useful Properties

- There are at most 2^l nodes at level (or depth) l of a binary tree
- A binary tree with maximum level d has at most 2^{d+1} - 1 nodes

$$n \leq \sum_{l=0}^{d} 2^l = \frac{2^{d+1}-1}{2-1} = 2^{d+1}-1$$



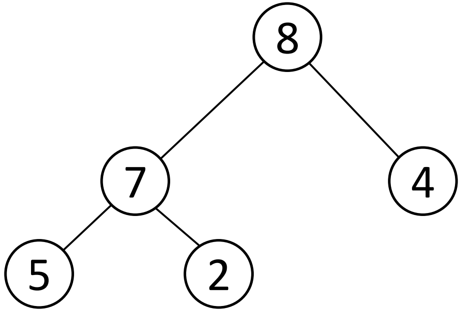
Height of root = 3

Height of (2)= 1

Level of (10)= 2

The Heap Data Structure

- *Def:* A **heap** is an almost complete binary tree with the following two properties:
  - **Structural property:** all levels are full, except possibly the last one, which is filled from left to right
  - **Max (Min) heap property:** for any node x, Parent(x) ≥ x (Parent(x)<=x)

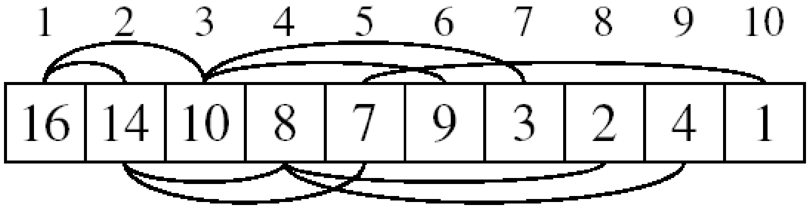


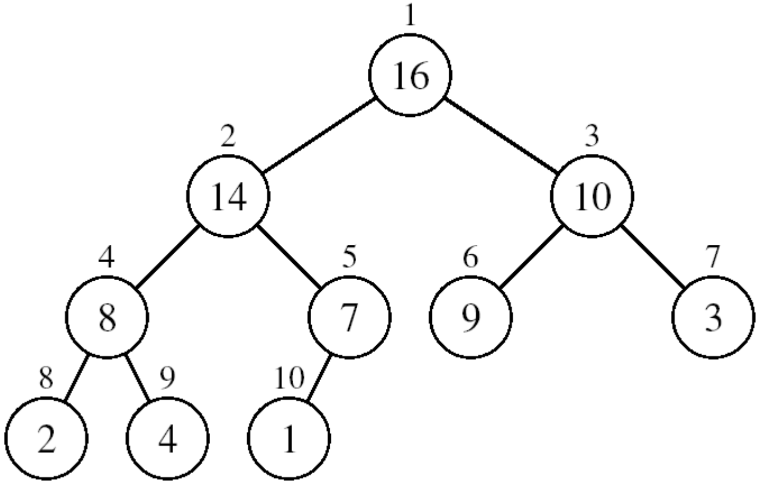
Max Heap

From the heap property, it follows that: The root is the maximum (minimum) element of the max-heap (min-heap)

Array Representation of Heaps

- A heap can be stored as an array A.
  - Root of tree is $A[1]$
  - Left child of $A[i] = A[2i]$
  - Right child of $A[i] = A[2i + 1]$
  - Parent of $A[i] = A[\lfloor i/2 \rfloor]$
  - Heapsize[A] ≤ length[A]
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1) .. n]$ are leaves





Heap Types

- **Max-heaps** (largest element at root), have the *max-heap property:*
  - for all nodes i, excluding the root:

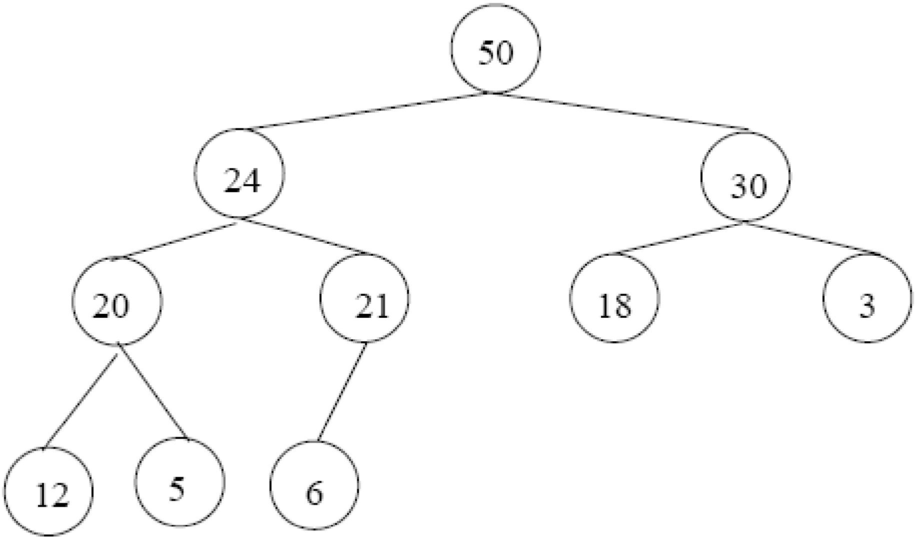
$$A[\text{PARENT}(i)] \geq A[i]$$

- **Min-heaps** (smallest element at root), have the *min-heap property:*
  - for all nodes i, excluding the root:

$$A[\text{PARENT}(i)] \leq A[i]$$

Adding/Deleting Nodes

- New nodes are always inserted at the bottom level (left to right)
- Nodes are removed from the bottom level (right to left)

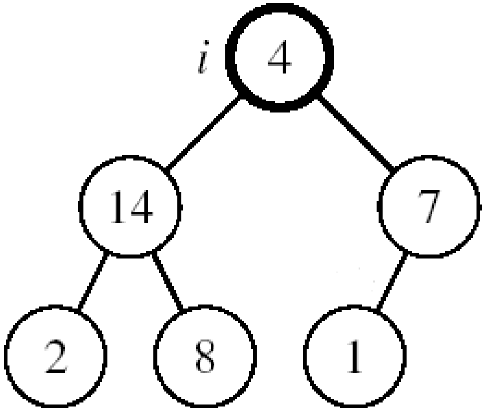


Operations on Heaps

- Maintain/Restore the max-heap property
    - MAX-HEAPIFY
- Create a max-heap from an unordered array
    - BUILD-MAX-HEAP
- Sort an array in place
    - HEAPSORT

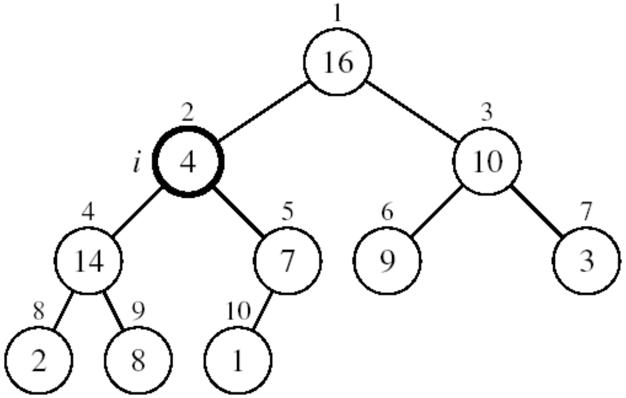
Maintaining the Heap Property

- Suppose a node is smaller than a child
  - Left and Right subtrees of i are max-heaps
- To eliminate the violation:
  - Exchange with larger child
  - Move down the tree
  - Continue until node is not smaller than children



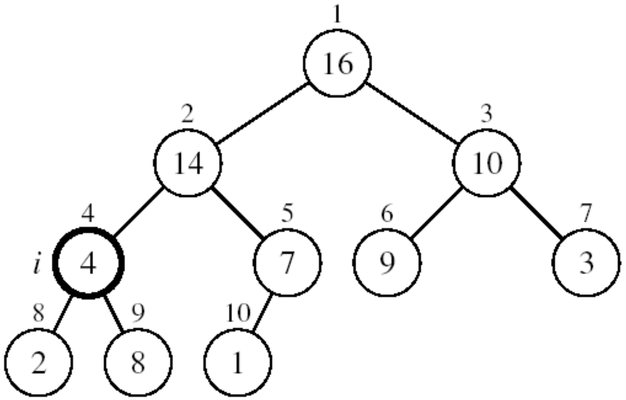
Example

MAX-HEAPIFY(A, 2, 10)



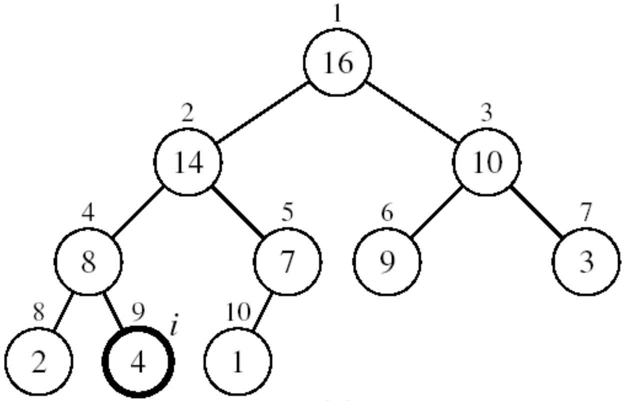
A[2] violates the heap property

A[2] ↔ A[4]



A[4] violates the heap property

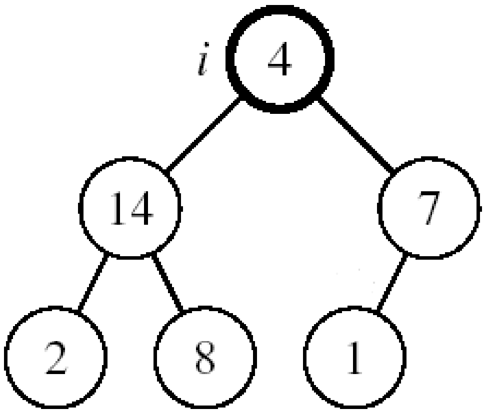
A[4] ↔ A[9]



Heap property restored

Maintaining the Heap Property

- Assumptions:
  - Left and Right subtrees of *i* are max-heaps
  - *A*[i] may be smaller than its children



Alg: MAX-HEAPIFY(*A*, i, n)

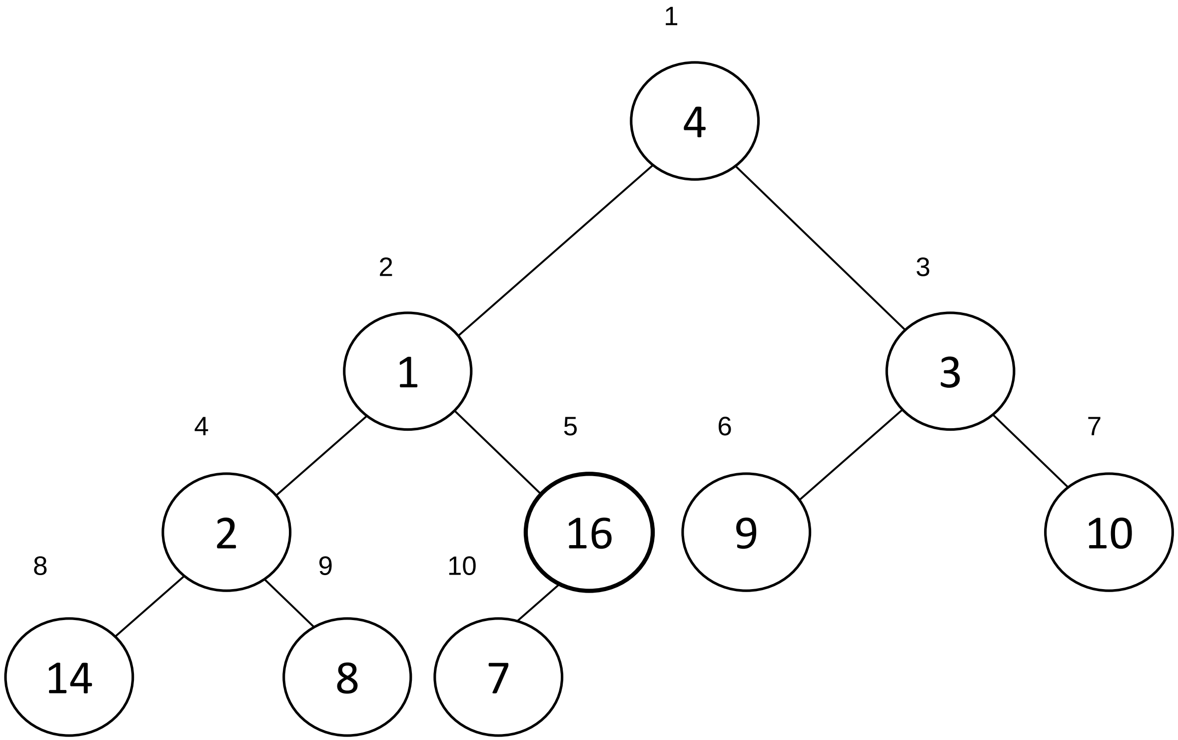
```
l ← LEFT(i)
r ← RIGHT(i)
if l ≤ n and A[l] > A[i]
    then largest ← l
    else largest ← i
if r ≤ n and A[r] > A[largest]
    then largest ← r
if largest ≠ i
    then exchange A[i] $\boxed{\leftrightarrow}$ A[largest]
            MAX-HEAPIFY(A, largest, n)
```

MAX-HEAPIFY Running Time

- It checks a path starting from current node to leaf node. At every level it performs exactly 2 comparisons. At max length of this path is h. So total number of comparisons is at most 2h. So complexity is O(h) or O(logn)

- Running time of MAX-HEAPIFY is $O(lgn)$

Building a Max-Heap

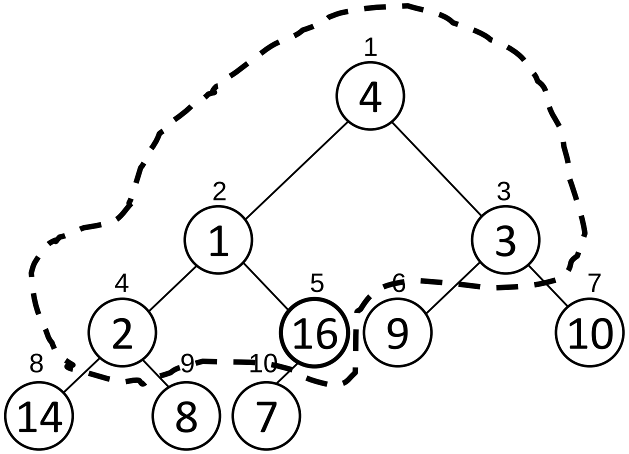


Building a Heap

- Convert an array $A[1 \ldots n]$ into a max-heap (n = length[A])
- The elements in the subarray $A[(\lfloor n/2 \rfloor+1) .. n]$ are leaves
- Apply MAX-HEAPIFY on elements between 1 and $\lfloor n/2 \rfloor$

Alg: BUILD-MAX-HEAP(A)

```
n = length[A]
for i ← ⌊n/2⌋ downto 1
    do MAX-HEAPIFY(A, i, n)
```



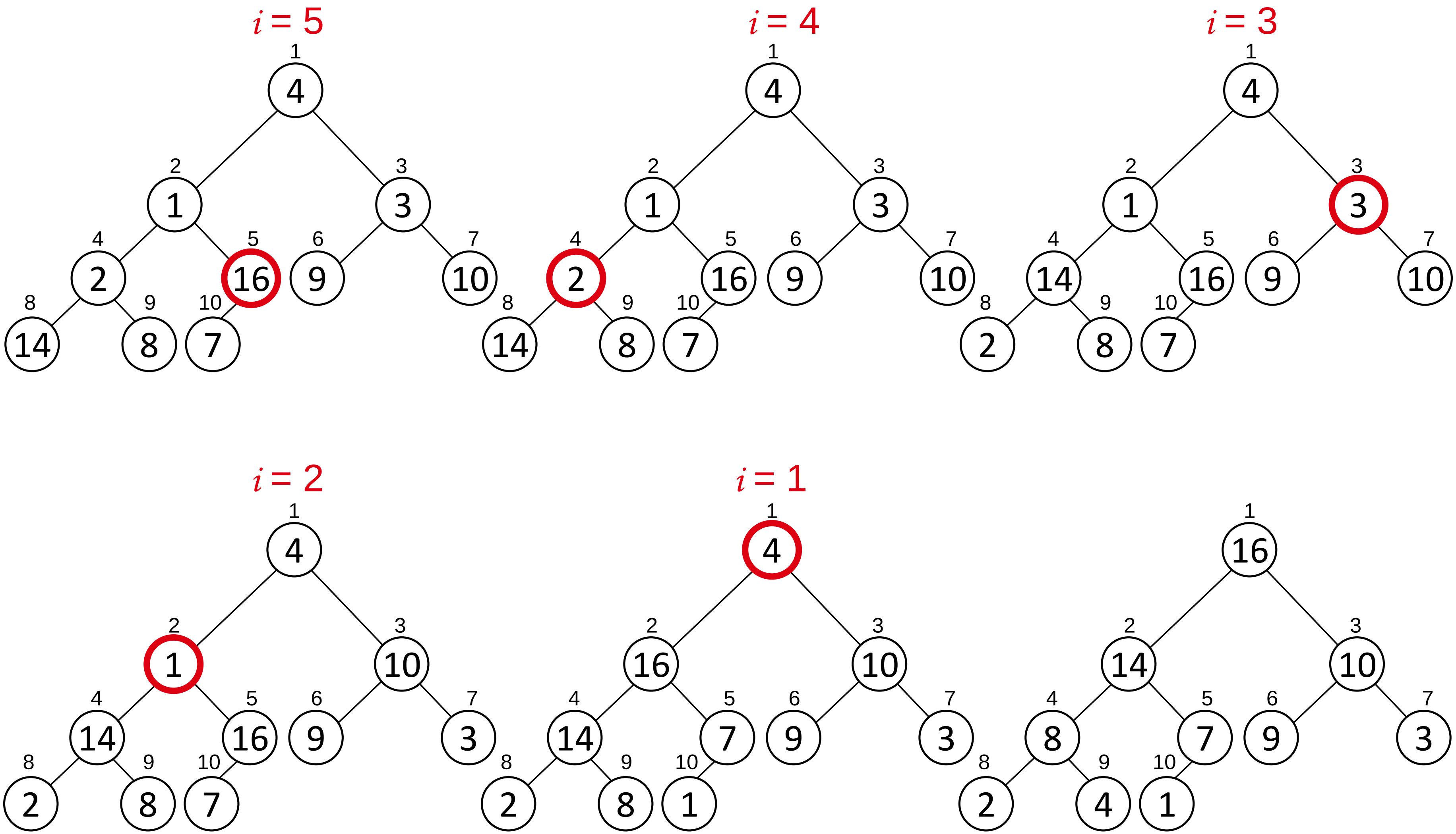
A:

4	1	3	2	16	9	10	14	8	7

Example:

A

4	1	3	2	16	9	10	14	8	7



Running Time of BUILD MAX HEAP

```
Alg: BUILD-MAX-HEAP(A)
n = length[A]
for i ← $\lfloor n/2 \rfloor$ downto 1                          } $O(n)$
    do MAX-HEAPIFY(A, i, n)                      $O(\lg n)$  }
```

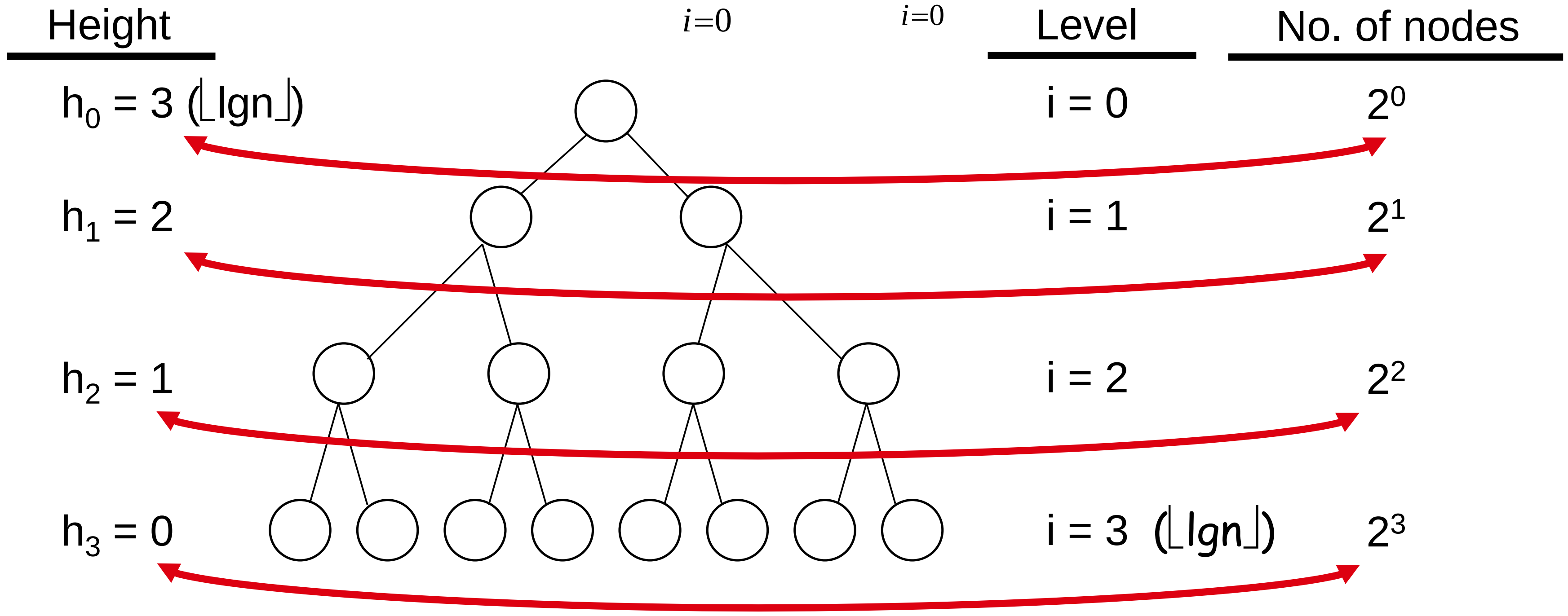
⇒ Running time: $O(n\lg n)$

- This is not an asymptotically tight upper bound

Running Time of BUILD MAX HEAP

- HEAPIFY takes $O(h)$ $\Rightarrow$ the cost of HEAPIFY on a node i is proportional to the height of the node i in the tree

$$\Rightarrow T(n) = \sum_{i=0}^{h} n_i h_i = \sum_{i=0}^{h} 2^i (h-i) = O(n)$$



Height	Level	No. of nodes
$h_0 = 3$ ($\lfloor \lg n \rfloor$)	$i = 0$	2^0
$h_1 = 2$	$i = 1$	2^1
$h_2 = 1$	$i = 2$	2^2
$h_3 = 0$	$i = 3$ ($\lfloor \lg n \rfloor$)	2^3

$h_i = h - i$ height of the heap rooted at level i

$n_i = 2^i$ number of nodes at level i

Running Time of BUILD MAX HEAP

$$T(n) = \sum_{i=0}^{h} n_i h_i$$

Cost of HEAPIFY at level i ∗ number of nodes at that level

$$= \sum_{i=0}^{h} 2^i (h - i)$$

Replace the values of n_i and h_i computed before

$$= \sum_{i=0}^{h} \frac{h - i}{2^{h-i}} 2^h$$

Multiply by 2^{h-i} both at the nominator and denominator

$$= 2^h \sum_{k=0}^{h} \frac{k}{2^k}$$

Change variables: k = h - i

$$\leq n \sum_{k=0}^{\infty} \frac{k}{2^k}$$

The sum above is smaller than the sum of all elements to ∞

$$= O(n)$$

The sum above is smaller than 2

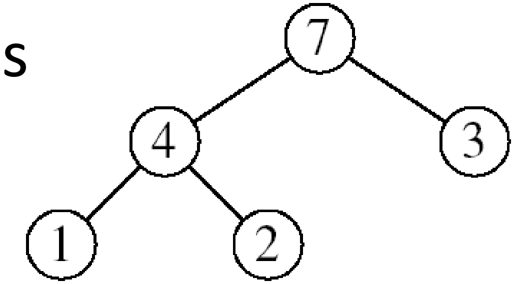
$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}$$

for $|x| < 1$.

Running time of BUILD-MAX-HEAP: **T(n) = O(n)**

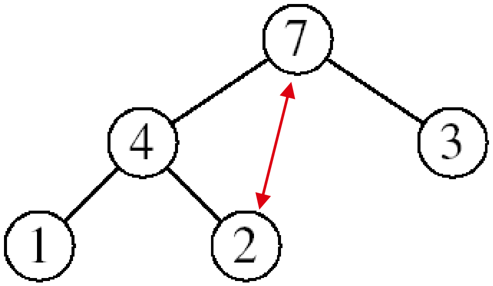
Heapsort

- Goal:
  - Sort an array using heap representations

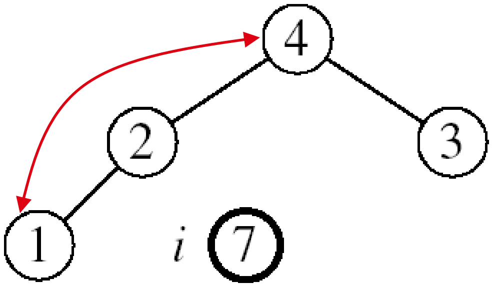


- Idea:
  - Build a **max-heap** from the array
  - Swap the root (the maximum element) with the last element in the array
  - “Discard” this last node by decreasing the heap size
  - Call MAX-HEAPIFY on the new root
  - Repeat this process until only one node remains

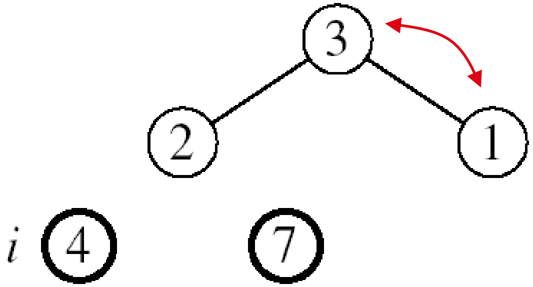
Example: A=[7, 4, 3, 1, 2]



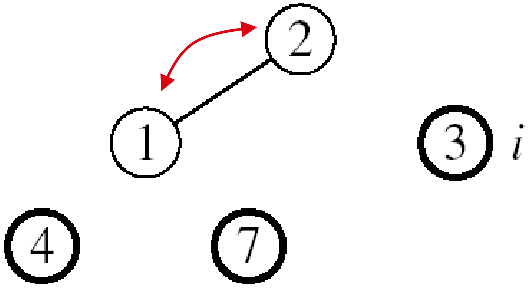
MAX-HEAPIFY(A, 1, 4)



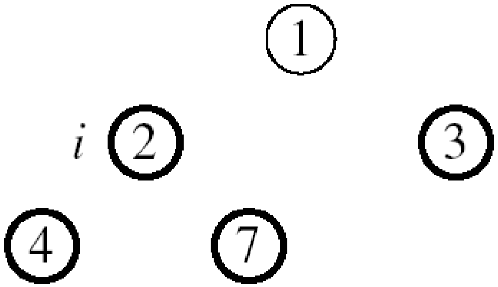
MAX-HEAPIFY(A, 1, 3)



MAX-HEAPIFY(A, 1, 2)



MAX-HEAPIFY(A, 1, 1)



A	1	2	3	4	7

Alg: HEAPSORT(*A*)

```
BUILD-MAX-HEAP(A)                              O(n)
for i ← length[A] downto 2
    do exchange A[1] ↔ A[i]
        MAX-HEAPIFY(A, 1, i - 1)               O(lgn)
```

$\Big\}$ n-1 times

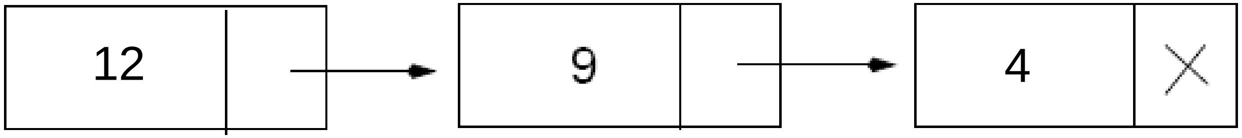
- Running time: O(nlgn) --- Can be shown to be Θ(nlgn)

Priority Queues

Properties

- Each element is associated with a value (priority)

- The key with the highest (or lowest) priority is extracted first



Operations
on Priority Queues

- Max-priority queues support the following operations:
  - INSERT(S, x): inserts element x into set S
  - EXTRACT-MAX(S): removes and returns element of S with largest key
  - MAXIMUM(S): returns element of S with largest key
  - INCREASE-KEY(S, x, k): increases value of element x's key to k (Assume $k \geq x$'s current key value)

HEAP-MAXIMUM

Goal:

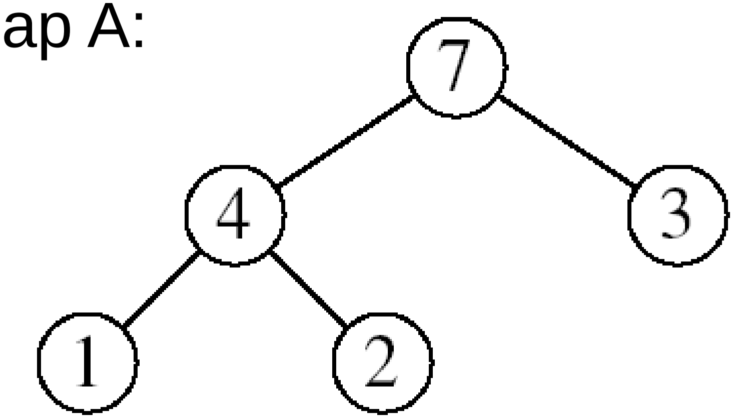
- Return the largest element of the heap

Running time: $O(1)$

Alg: HEAP-MAXIMUM(A)

```
1.    return $A[1]$
```

Heap A:



Heap-Maximum(A) returns 7

HEAP-EXTRACT-MAX

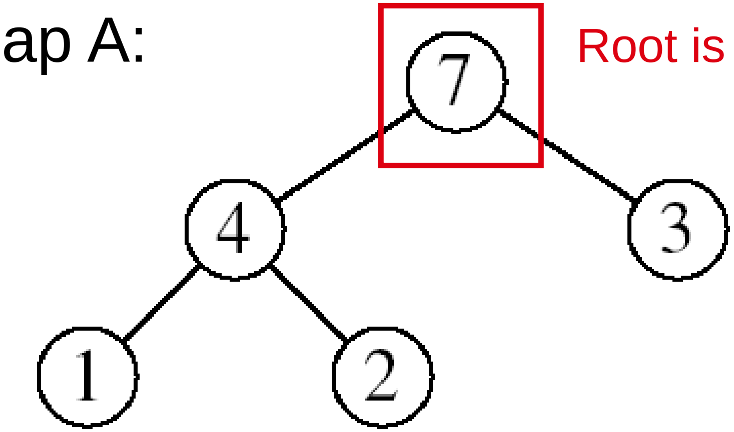
Goal:

- Extract the largest element of the heap (i.e., return the max value and also remove that element from the heap

Idea:

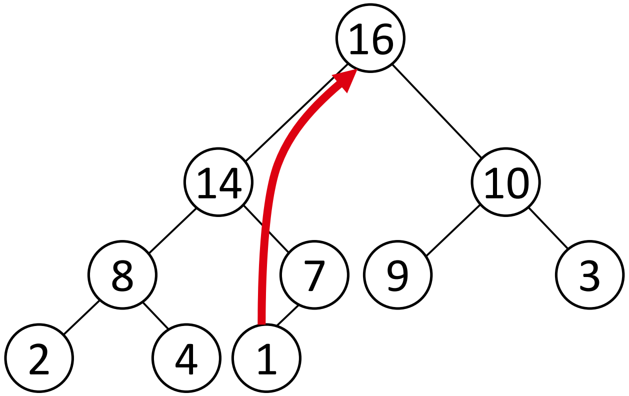
- Exchange the root element with the last
- Decrease the size of the heap by 1 element
- Call MAX-HEAPIFY on the new root, on a heap of size n-1

Heap A:

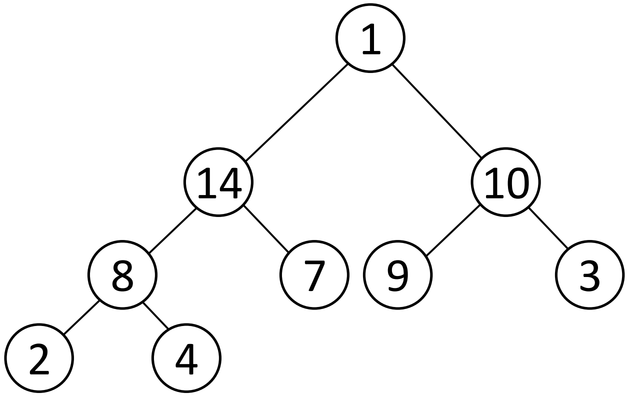


Root is the largest element

Example: HEAP-EXTRACT-MAX

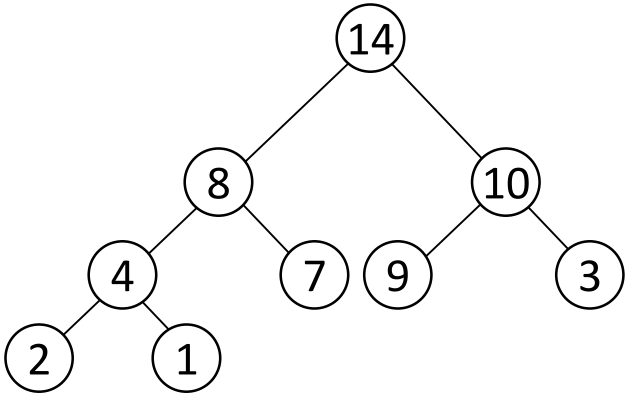


max = 16



Heap size decreased with 1

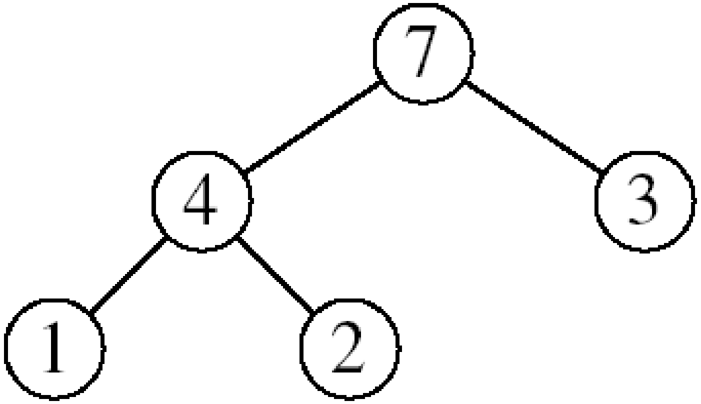
Call MAX-HEAPIFY(A, 1, n-1)



HEAP-EXTRACT-MAX

Alg: HEAP-EXTRACT-MAX(*A*, *n*)

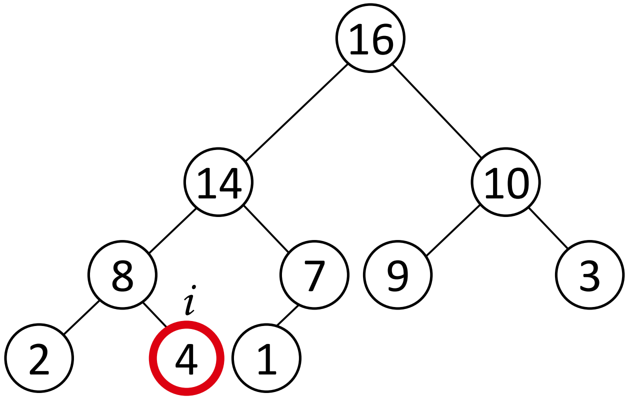
```
if $n < 1$
    then error "heap underflow"
$max \leftarrow A[1]$
$A[1] \leftarrow A[n]$
MAX-HEAPIFY($A$, 1, $n$-1)          ▷ remakes heap
return $max$
```



Running time: $O(\lg n)$

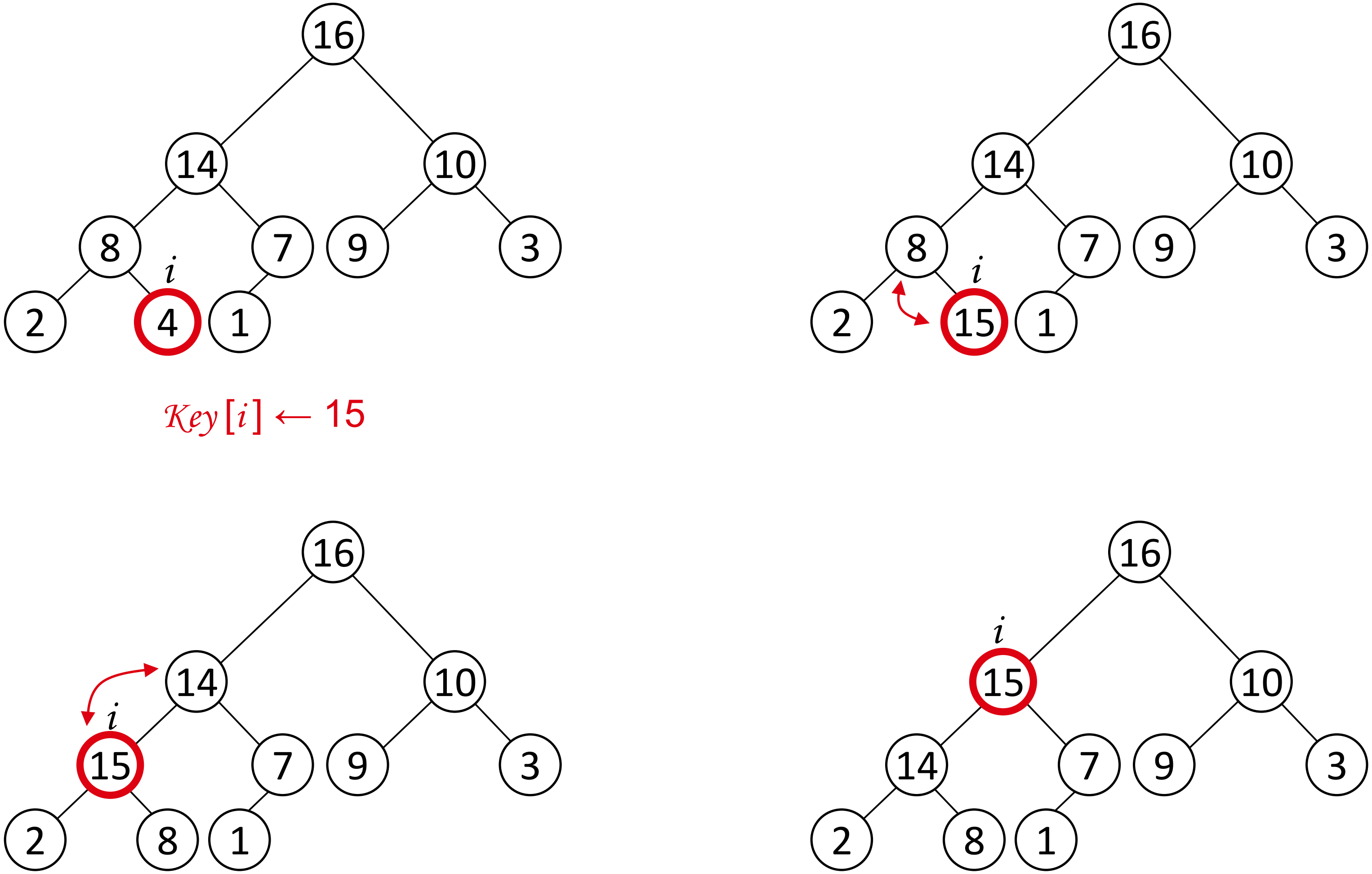
HEAP-INCREASE-KEY

- Goal:
  - Increases the key of an element i in the heap
- Idea:
  - Increment the key of $A[i]$ to its new value
  - If the max-heap property does not hold anymore: traverse a path toward the root to find the proper place for the newly increased key



Key [i] ← 15

Example: HEAP-INCREASE-KEY



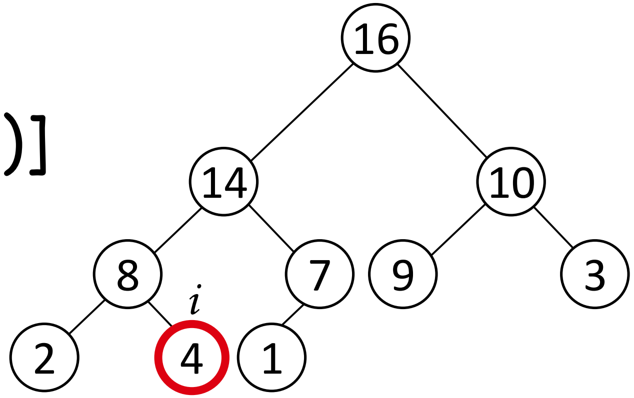
$Key[i] \leftarrow 15$

HEAP-INCREASE-KEY

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

```
if key < A[i]
   then error "new key is smaller than current key"
A[i] ← key
while i > 1 and A[PARENT(i)] < A[i]
    do exchange A[i] ↔ A[PARENT(i)]
        i ← PARENT(i)
```

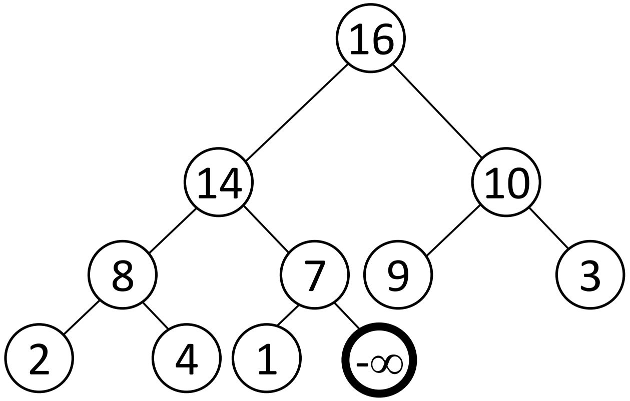
- Running time: $O(\lg n)$

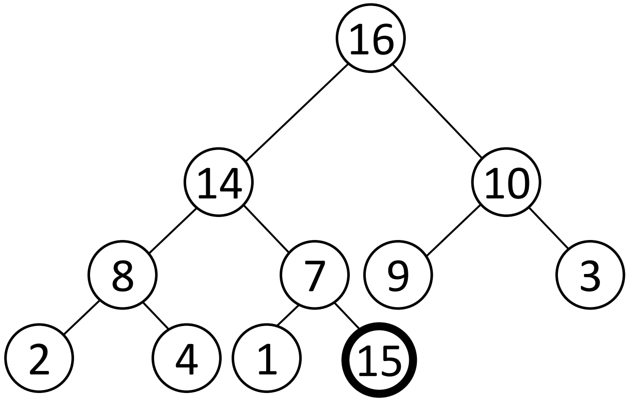


Key [i] ← 15

MAX-HEAP-INSERT

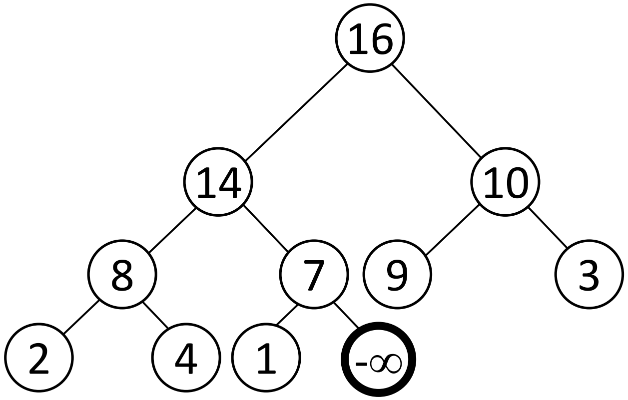
- Goal:
  - Inserts a new element into a max-heap
- Idea:
  - Expand the max-heap with a new element whose key is -∞
  - Calls HEAP-INCREASE-KEY to set the key of the new node to its correct value and maintain the max-heap property



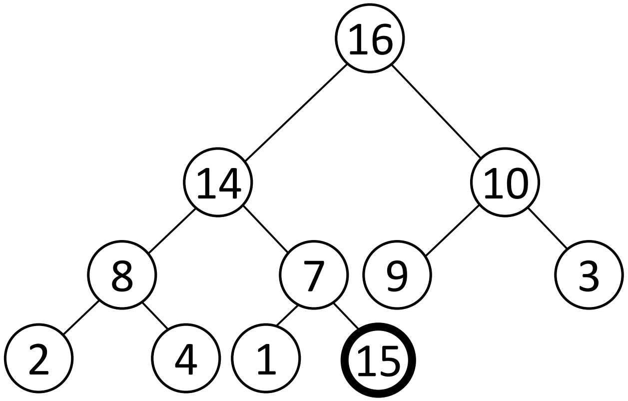


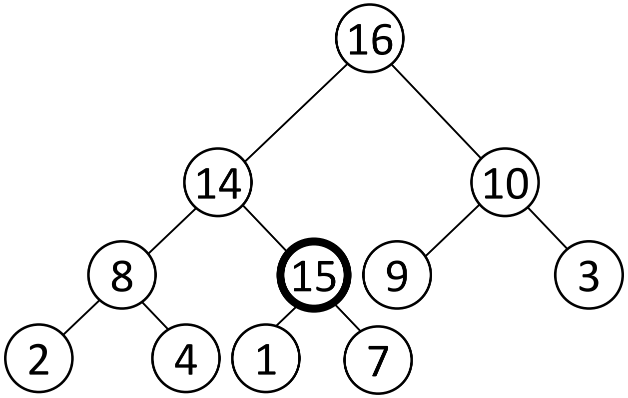
Example: MAX-HEAP-INSERT

Insert value 15:
- Start by inserting -∞

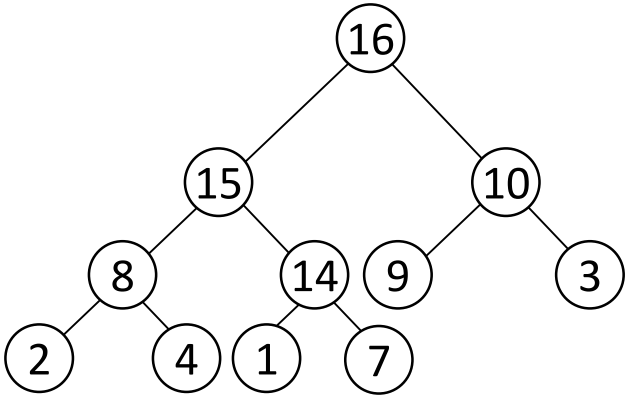


Increase the key to 15
Call HEAP-INCREASE-KEY on A[11] = 15



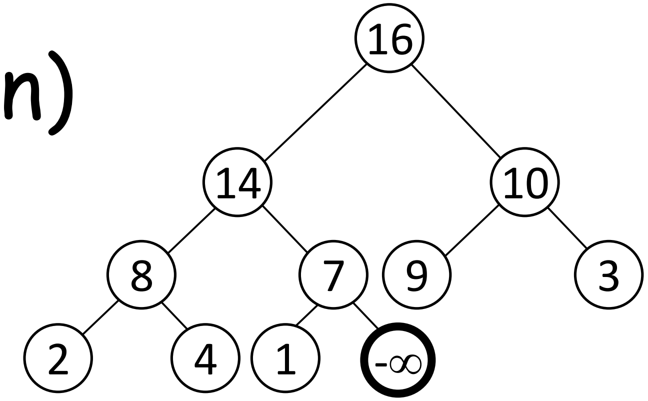


The restored heap containing
the newly added element



MAX-HEAP-INSERT

```
Alg: MAX-HEAP-INSERT($A$, key, $n$)
    heap-size[$A$] $\leftarrow n + 1$
    $A[n + 1] \leftarrow -\infty$
    HEAP-INCREASE-KEY($A$, $n + 1$, key)
```



Running time: $O(\lg n)$

Summary

- We can perform the following operations on heaps:

– MAX-HEAPIFY	$O(lgn)$
– BUILD-MAX-HEAP	$O(n)$
– HEAP-SORT	$O(nlgn)$
– MAX-HEAP-INSERT	$O(lgn)$
– HEAP-EXTRACT-MAX	$O(lgn)$
– HEAP-INCREASE-KEY	$O(lgn)$
– HEAP-MAXIMUM	$O(1)$

Average $O(lgn)$

Quick Sort

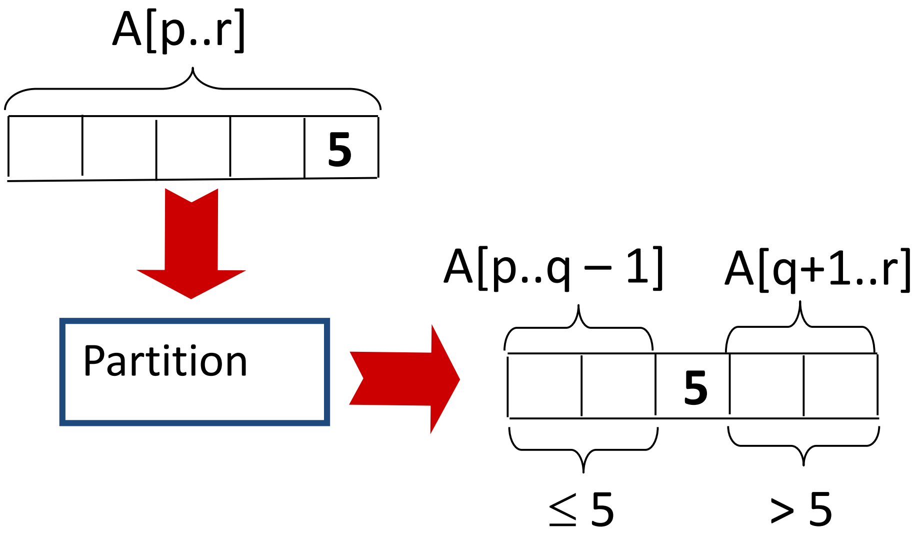
QuickSort Design

- Follows the **divide-and-conquer** paradigm.
- ***Divide***: Partition (separate) the array $A[p..r]$ into two (possibly empty) subarrays $A[p..q-1]$ and $A[q+1..r]$.
  - Each element in $A[p..q-1] <= A[q]$.
  - $A[q] <$ each element in $A[q+1..r]$.
  - Index q is computed as part of the partitioning procedure.
- ***Conquer***: Sort the two subarrays by recursive calls to quicksort.

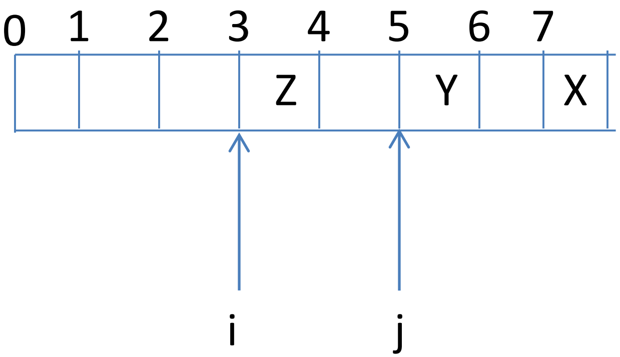
- ***Combine***: The subarrays are sorted in place – no work is needed to combine them.
- How do the divide and conquer steps of quicksort compare with those of merge sort?

Pseudocode

```
Quicksort(A, p, r)
    if p < r then
        q := Partition(A, p, r);
        Quicksort(A, p, q – 1);
        Quicksort(A, q + 1, r)
```



```
Partition(A, p, r)
    x:= A[r]
    i=p – 1;
    for j := p to r – 1 do
        if A[j] ≤ x then
            i := i + 1;
            A[i] ↔ A[j]
    A[i + 1] ↔ A[r];
    return i + 1
```



Example

Position of i and j after of line 3

p r

initially: 2 5 8 3 9 4 1 7 10 **6**

i j

note: pivot (x) = 6

next iteration: 2 5 8 3 9 4 1 7 10 **6**

i j

next iteration: 2 5 8 3 9 4 1 7 10 **6**

i j

next iteration: 2 5 8 3 9 4 1 7 10 **6**

i j

next iteration: 2 5 3 8 9 4 1 7 10 **6**

i j

```
Partition(A, p, r)
1   x:= A[r]
2    i=p – 1;
3   for j := p to r – 1 do
4           if A[j] ≤ x then
5                   i := i + 1;
6                   A[i] ↔ A[j]
7   A[i + 1] ↔ A[r];
8   return i + 1
```

Example (Continued)

next iteration:	2	5	3	8	9	4	1	7	10	**6**
			i			j				
next iteration:	2	5	3	4	9	8	1	7	10	**6**
				i			j			
next iteration:	2	5	3	4	1	8	9	7	10	**6**
					i			j		
next iteration:	2	5	3	4	1	8	9	7	10	**6**
					i				j	
after final swap:	2	5	3	4	1	**6**	9	7	10	8
						i				j

```
Partition(A, p, r)
    x:= A[r]
    i=p – 1;
    for j := p to r – 1 do
        if A[j] $\leq$ x then
            i := i + 1;
            A[i] $\leftrightarrow$ A[j]
    A[i + 1] $\leftrightarrow$ A[r];
    return i + 1
```

Partitioning

- Select the last element A[*r*] in the subarray *A*[*p*..*r*] as the *pivot* – the element around which to partition.
- As the procedure executes, the array is partitioned into four (possibly empty) regions.
  1. *A*[*p*..*i*] — All entries in this region are **<= *pivot***.
  2. *A*[*i*+1..*j* – 1] — All entries in this region are **> *pivot***.
  3. *A*[*r*] = *pivot*.
  4. *A*[*j*..*r* – 1] — Not known how they compare to *pivot*.
- The above hold before each iteration of the *for* loop, and constitute a *loop invariant.*

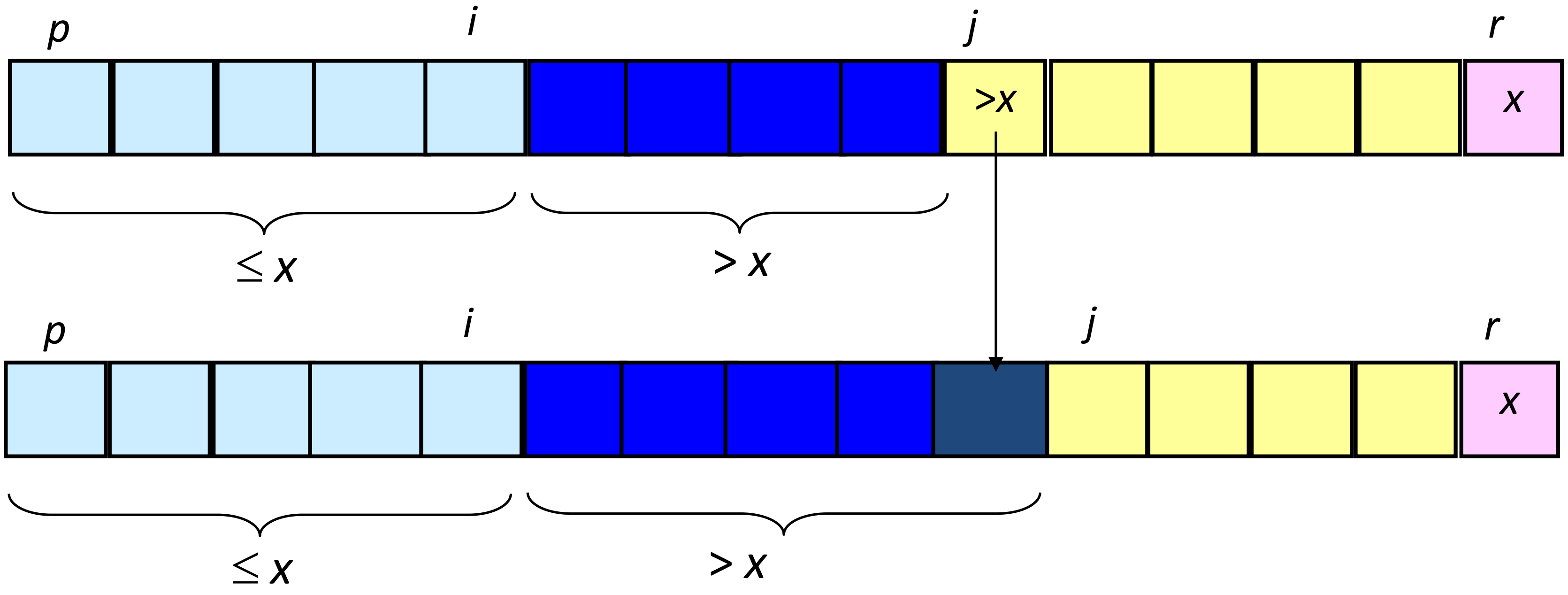
Correctness of Partition

- Use loop invariant.
- **Initialization:**
  - Before first iteration
    - $A[p..i]$ and $A[i+1..j-1]$ are empty – Conds. 1 and 2 are satisfied (trivially).
    - r is the index of the *pivot*
      - Cond. 3 is satisfied.
- **Maintenance:**
  - **Case 1:** $A[j] > x$
    - Increment j only.
    - Loop Invariant is maintained.

```
Partition(A, p, r)
        x:= A[r]
        i=p – 1;
        for j := p to r – 1 do
                if A[j] ≤ x then
                        i := i + 1;
                        A[i] ↔ A[j]
        A[i + 1] ↔ A[r];
        return i + 1
```

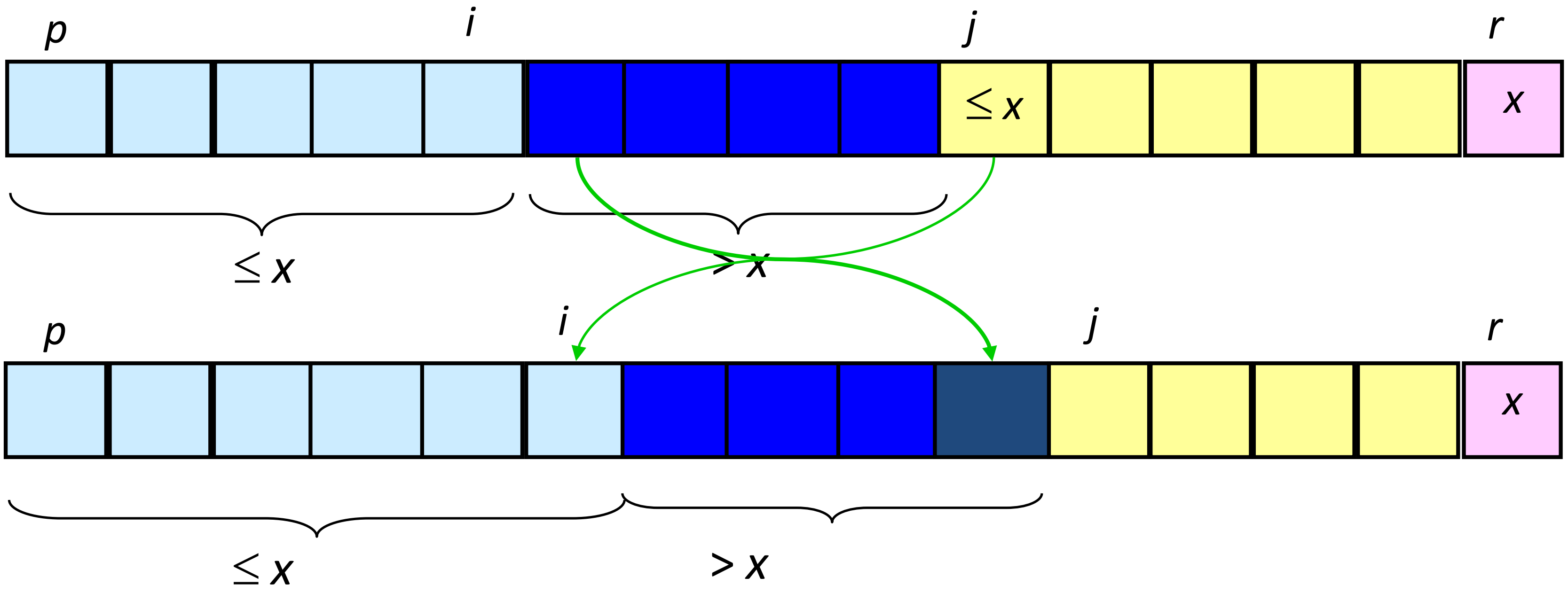
Correctness of Partition

Case 1:



Correctness of Partition

- **Case 2:** $A[j] \le x$
  - Increment i
  - Swap $A[i]$ and $A[j]$
    - Condition 1 is maintained.
  - Increment j
    - Condition 2 is maintained.
  - $A[r]$ is unaltered.
    - Condition 3 is maintained.



Correctness of Partition

- **Termination:**
  - When the loop terminates, $j = r$, so all elements in A are partitioned into one of the three cases:
    - $A[p..i] \leq$ ***pivot***
    - $A[i+1..j - 1] >$ ***pivot***
    - $A[r] =$ ***pivot***
- The last two lines swap $A[i+1]$ and $A[r]$.
  - *Pivot* moves from the end of the array to between the two subarrays.
  - Thus, procedure *partition* correctly performs the divide step.

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$
  - Number of accesses in partition?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- What is best case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays of size n/2
    2. Quicksort each sub-array
  - Depth of recursion tree? $O(\log_2 n)$
  - Number of accesses in partition? $O(n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Recurrence relation $T(n)=2T(n/2)+\Theta(n)$
- Running time: $O(n \log_2 n)$

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: O(n $\log_2 n$)
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree?

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Best case running time: $O(n \log_2 n)$
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)
  - Number of accesses per partition?

Quicksort Analysis

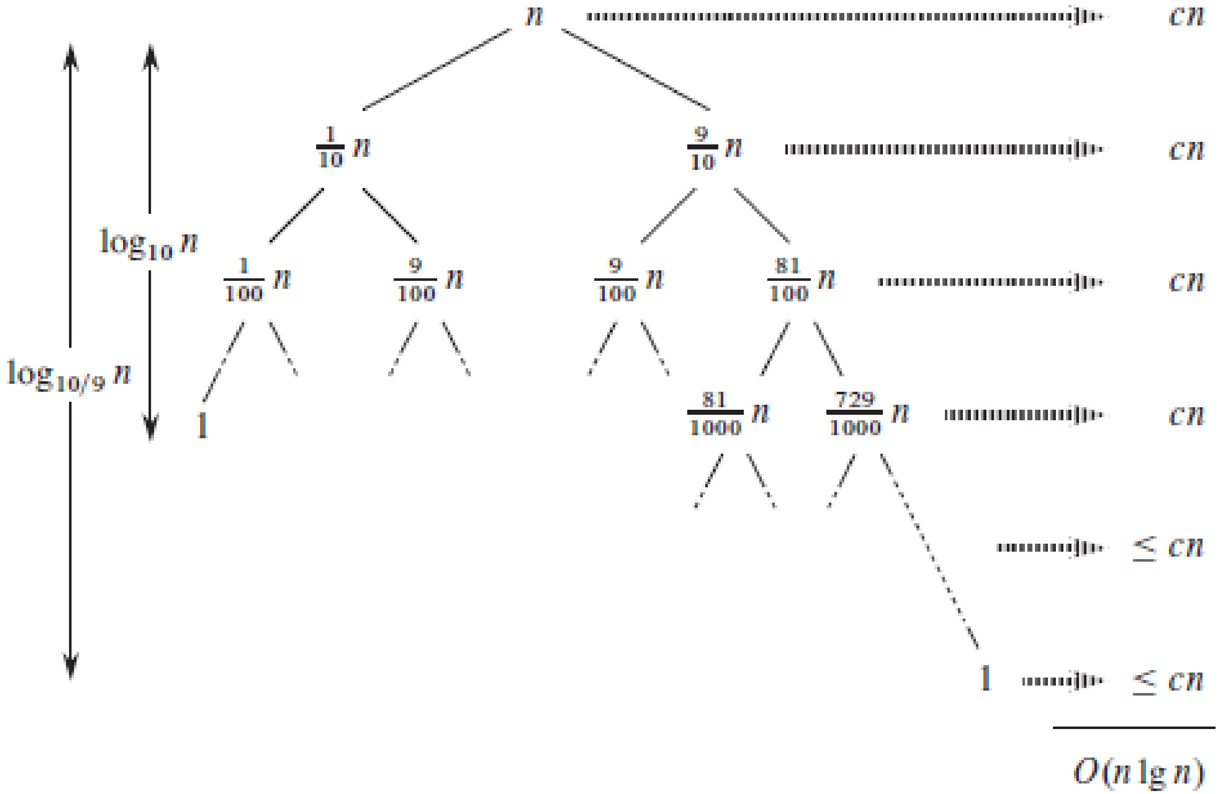
- Assume that keys are random, uniformly distributed.
- Best case running time: O(n $\log_2 n$)
- Worst case running time?
  - Recursion:
    1. Partition splits array in two sub-arrays:
       - one sub-array of size 0
       - the other sub-array of size n-1
    2. Quicksort each sub-array
  - Depth of recursion tree? O(n)
  - Number of accesses per partition? O(n)

Quicksort Analysis

- Assume that keys are random, uniformly distributed.
- Recurrence Relation $T(n)=T(n-1)+\Theta(n)$
- Worst case running time: $O(n^2)$

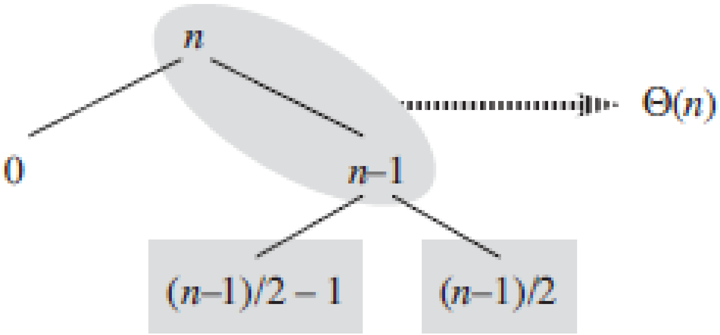
Quicksort Analysis

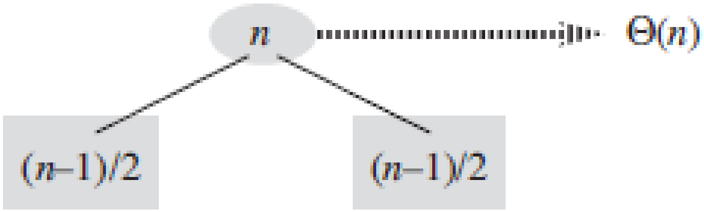
- **Balanced partitioning**



Quicksort Analysis

- mix of “good” and “bad” splits





A randomized version of quicksort

- RANDOMIZED-PARTITION (A, p, r)

```
i = RANDOM(p, r)
exchange A[r] with A[i]
return PARTITION(A, p, r)
```

- RANDOMIZED-QUICKSORT(A,p,r)

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

Expected running time

- The running time of QUICKSORT is dominated by the time spent in the PARTITION procedure.
- Each time the PARTITION procedure is called, it selects a pivot element, and this element is never included in any future recursive calls to QUICKSORT and PARTITION.
- Thus, there can be at most n calls to PARTITION over the entire execution of the quicksort algorithm.

Expected Running Time of Partition

- One call to PARTITION takes O(1) time plus an amount of time that is proportional to the number of iterations of the **for** loop in lines 3–6

if we can count the total number of times that line 4 is executed, we can bound the total time spent in the for loop during the entire execution of QUICKSORT.

```
Partition(A, p, r)
1.  x:= A[r],
2.          i=p – 1;
3.  for j := p to r – 1 do
4.          if A[j] ≤ x then
5.                  i := i + 1;
6.                  A[i] ↔ A[j]
7.  A[i + 1] ↔ A[r];
8.  return i + 1
```

- Assume line 4 is executed for X times
- We rename the elements of the array A as z_1, z_2, … , z_n, with z_i being the i^{th} smallest element.

- We also define the set $Z_{ij} = \{z_i , z_{i+1} , …, z_j \}$ to be the set of elements between z_i and z_j , inclusive.

When does the algorithm compare z_i and z_j ?

- X_{ij} = 1 {z_i is compared to z_j};
- X=$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}X_{ij}$
- E[X]=E[$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}X_{ij}$]

  =$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}E[Xij]$

  =$\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}\mathrm{Prob}$\{zi is compared to zj\}

- once a pivot x is chosen with $z_i < x < z_j$, we know that z_i and z_j will not be compared at any subsequent time.
- Pr {z_i is compared to z_j } = Pr {z_i or z_j is first pivot chosen from Z_{ij} }

= Pr {z_i is the first pivot chosen from Z_{ij} } + Pr {z_j is the first pivot chosen from Z_{ij} }

=1/(j-i+1)+ 1/(j-i+1)= 2/(j-i+1)

$$
\begin{aligned}
\mathrm{E}[X] &= \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \\
&= \sum_{i=1}^{n-1} \sum_{k=1}^{n-i} \frac{2}{k+1} \\
&< \sum_{i=1}^{n-1} \sum_{k=1}^{n} \frac{2}{k} \\
&= \sum_{i=1}^{n-1} O(\lg n) \\
&= O(n \lg n) .
\end{aligned}
$$

Linear Sorts

Counting sort

Bucket sort

Radix sort

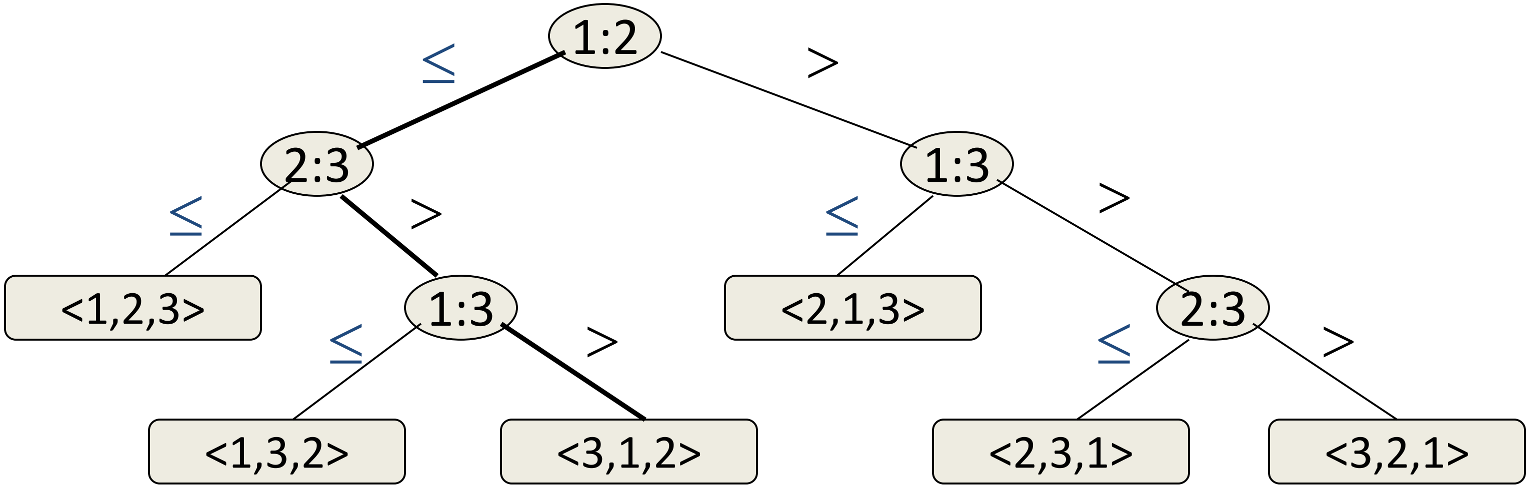
Comparison Sorting

- Given a set of *n* values, there can be *n!* permutations of these values.
- So if we look at the behavior of the sorting algorithm over all possible *n!* inputs we can determine the worst-case complexity of the algorithm.

Decision Tree

- Decision tree model
  - Full binary tree
  - Internal node represents a comparison.
  - Each leaf represents one possible result (a permutation of the elements in sorted order).
  - The height of the tree (i.e., longest path) is the lower bound.

Decision Tree Model



Internal node *i*:*j* indicates comparison between a_i and a_j.
suppose three elements < *a*1, *a*2, *a*3> with instance <6,8,5>
Leaf node <π(1), π(2), π(3)> indicates ordering $a_{\pi(1)} \leq a_{\pi(2)} \leq a_{\pi(3)}$.
Path of **bold lines** indicates sorting path for <6,8,5>.
There are total 3!=6 possible permutations (paths).

Decision Tree Model

- The longest path is the worst case number of comparisons. The length of the longest path is the height of the decision tree.
- **Theorem 8.1**: Any comparison sort algorithm requires $\Omega(n\lg n)$ comparisons in the worst case.
- **Proof:**
  - Suppose height of a decision tree is h, and number of paths (i,e,, permutations) is $n!$.
  - Since a binary tree of height h has at most 2^h leaves,
    - $n! \leq 2^h$, so $h \geq \lg (n!)$

Decision Tree Model

$$n! = \sqrt{2\pi n}\left(\frac{n}{e}\right)^n\left(1+\Theta\left(\frac{1}{n}\right)\right)$$

$$\lg(n!) \quad = \quad \Theta(n\lg n)\ ,$$

- That is to say: **any comparison sort in the worst case needs at least *n*lg *n* comparisons.**

Linear Sorts

- We will study algorithms that do not depend only on comparing *whole* keys to be sorted.

- Counting sort
- Bucket sort
- Radix sort

Counting sort

- **Assumptions:**
  - *n* records
  - Each record has a key and a value
  - All keys are in the range of 1 to k
- **Space**
  - The unsorted list is stored in A, the sorted list will be stored in an additional array B
  - Uses an additional array C of size k

Counting sort

- **Main idea:**

  1. For each key value *i*, *i* = 1,…,*k,* count the number of times the keys occurs in the unsorted input array *A.*

     Store results in an auxiliary array, C

  2. Use these counts to compute the offset. Offset$_i$ is used to calculate the location where the record with key value *i* will be stored in the sorted output list *B.*

     The offset$_i$ value has the location where the last key$_\mathrm{i}$.

- **When would you use counting sort?**
- **How much memory is needed?**

Counting Sort

Input: $A[1 .. n]$,
$A[J] \in \{1,2, . . . , k\}$

Output: $B[1 .. n]$,
sorted

Uses $C[1 .. k]$,
auxiliary storage

```
Counting-Sort( $A, B, k$)
    for $i \leftarrow 1$ to $k$
        do    $C[i] \leftarrow 0$
    for $j \leftarrow 1$ to $length[A]$
        do    $C[A[j]] \leftarrow C[A[j]] + 1$
    for $i \leftarrow 2$ to $k$
        do    $C[i] \leftarrow C[i] + C[i-1]$
    for $j \leftarrow length[A]$ down 1
        do    $B[C[A[j]]] \leftarrow A[j]$
              $C[A[j]] \leftarrow C[A[j]] - 1$
```

Analysis:

Adapted from Cormen,Leiserson,Rivest

	1	2	3	4	5	6
A	4	1	3	4	3	4

k = 4, *length* = 6

```
Counting-Sort( $A$, $B$, $k$)
    1. for $i \leftarrow 1$ to $k$
    2.     do      $C[i] \leftarrow 0$
    3. for $j \leftarrow 1$ to $length[A]$
    4.     do      $C[A[j]] \leftarrow C[A[j]] + 1$
    5. for $i \leftarrow 2$ to $k$
    6.     do      $C[i] \leftarrow C[i] + C[i-1]$
```

C	0	0	0	0

after lines 1-2

C	1	0	2	3

after lines 3-4

C	1	1	3	6

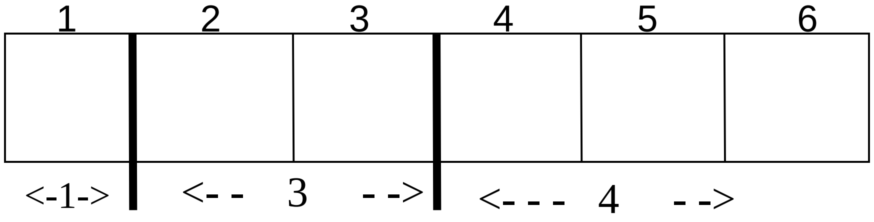
after lines 5-6

A

1	2	3	4	5	6
4	1	3	4	3	4

```
for $j \leftarrow length[A]$ down 1
    do   $B[C[A[j]]] \leftarrow A[j]$
         $C[A[j]] \leftarrow C[A[j]] - 1$
```

B



C

1	1	3	6

A

1	2	3	4	5	6
4	1	3	4	3	4

```
for j ← length[A] down 1
    do B [ C[A[ j ] ] ] ← A[ j ]
       C[A[ j ] ] ] ← C [A[ j ] ] -1
```

B

1	2	3	4	5	6
					4

<-1-> <-- 3 --> <--- 4 -->

J=6

C

1	1	3	5

B

1	2	3	4	5	6
		3			4

<-1-> <-- 3 --> <--- 4 -->

J=5

C

1	1	2	5

	1	2	3	4	5	6
A	4	1	3	4	3	4

```
7. for  j ← length[A] down 1
8.     do        B [ C[A[ j ] ] ] ← A[ j ]
9.               C[A[ j ] ] ] ← C [A[ j ] ] -1
```

	1	2	3	4	5	6
B	1		3		4	4
	<-1->	<-- 3	-->	<--- 4		-->

J=4

C	1	1	2	4

	1	2	3	4	5	6
B		3	3		4	4
	<-1->	<-- 3	-->	<--- 4		-->

J=3

C	1	1	1	4

A

1	2	3	4	5	6
4	1	3	4	3	4

```
for j ← length[A] down 1
    do B[ C[A[ j ] ] ] ← A[ j ]
       C[A[ j ] ] ] ← C [A[ j ] ] -1
```

B

1	2	3	4	5	6
1	3	3		4	4

<-1-> <-- 3 --> <--- 4 -->

J=2

C

0	1	1	4

B

1	2	3	4	5	6
1	3	3	4	4	4

<-1-> <-- 3 --> <--- 4 -->

J=1

C

0	1	1	3

Analysis:

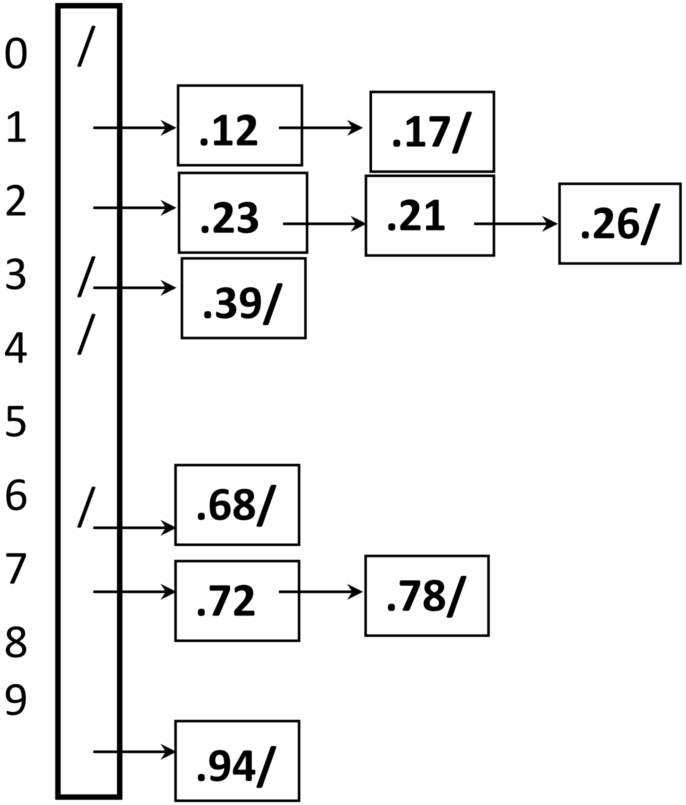
- O(k + n) time
  - What if k = O(n)
- But Sorting takes Ω (n lg n) ????
- Requires k + n extra storage.
- This is a stable sort: It preserves the original order of equal keys.
- Clearly no good for sorting 32 bit values.

Bucket sort

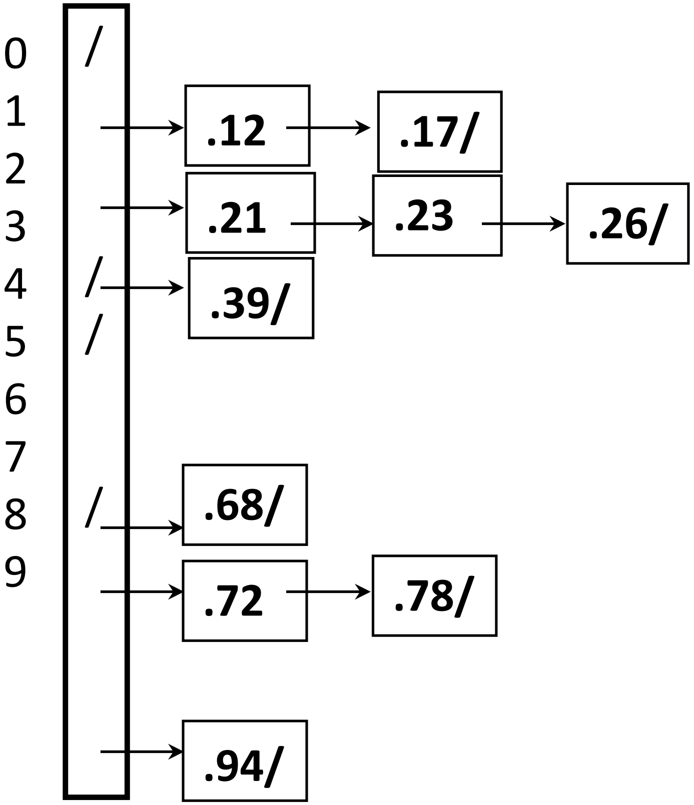
- Keys are distributed uniformly in interval [0, 1)
- The records are distributed into n buckets
- The buckets are sorted using one of the well known sorts
- Finally the buckets are combined

Bucket sort

1	.78
2	.17
3	.39
4	.26
5	.72
6	.94
7	.21
8	.12
9	.23
10	.68



Step 1 distribute



Step 2 sorted

Step3 combine

Analysis

- $P = 1/n$, probability that the key goes to bucket i.
- Expected size of bucket is $np = n * 1/n = 1$

- The expected time to sort one bucket is $\Theta(1)$.

- Overall expected time is $\Theta(n)$.

Radix sort

- Main idea
  - Break key into “digit” representation

    key = $i_d, i_{d-1}, \ldots, i_2, i_1$
  - "digit" can be a number in any base, a character, etc
- Radix sort:

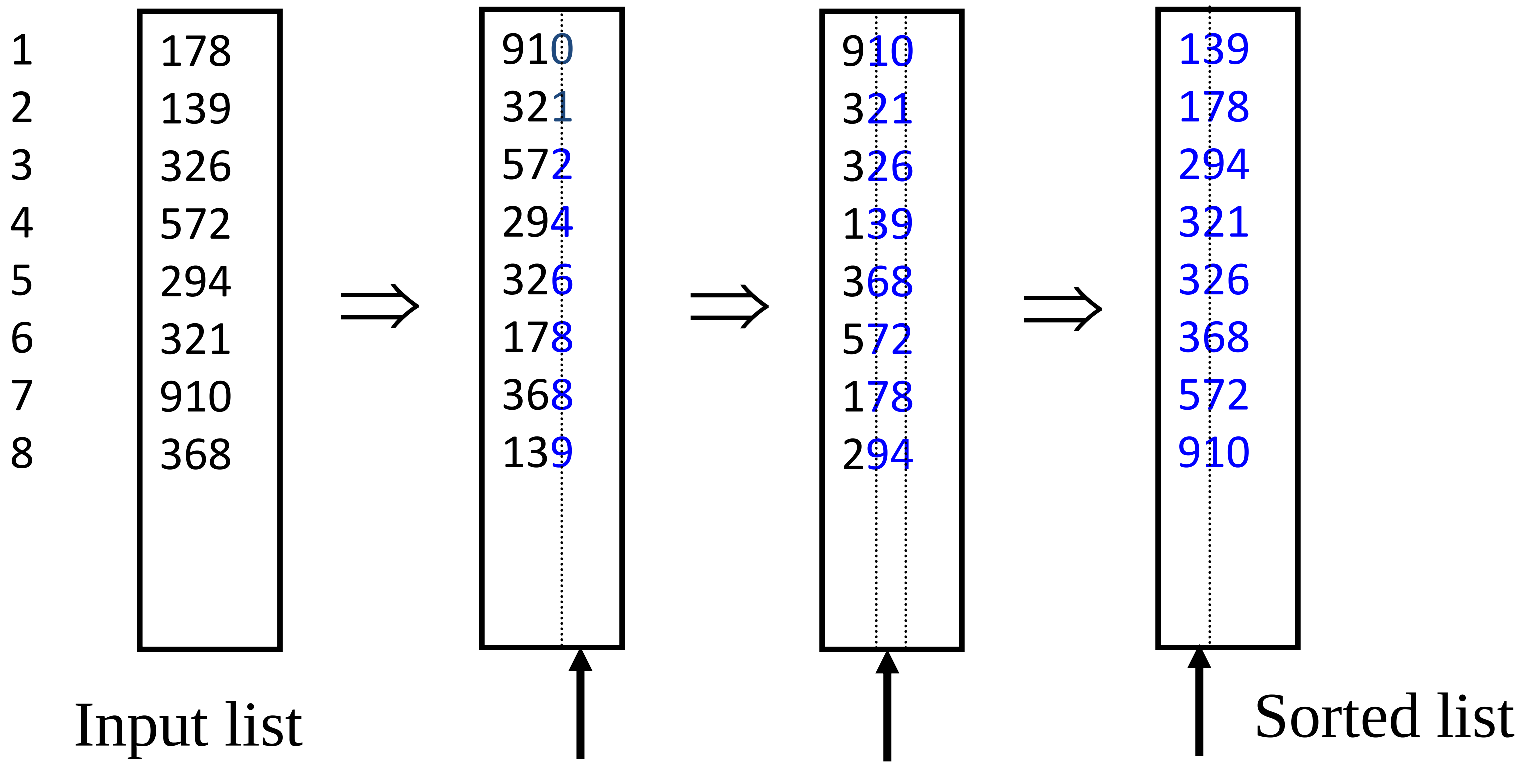
```
for i= 1 to d
    sort “digit” i using a stable sort
```

- Analysis : Θ(d $*$ (stable sort time)) where d is the number of “digit”s

Radix sort

- Which stable sort?
  - Since the range of values of a digit is small the best stable sort to use is Counting Sort.
  - When counting sort is used the time complexity is $\Theta(d * (n + k))$ where k is the range of a "digit".
    - When $k \in O(n)$, $\Theta(d * n)$

Radix sort- with decimal digits



1	178	910	910	139
2	139	321	321	178
3	326	572	326	294
4	572	294	139	321
5	294	326	368	326
6	321	178	572	368
7	910	368	178	572
8	368	139	294	910

Input list

Sorted list