ENPM 809X

E. Gunduzhan

Module 3: Probabilistic analysis; Heap sort

Probabilistic Analysis and Randomized Algorithms

Hiring Problem

- Suppose you always want to have the best qualified person as your office assistant, and have ongoing daily interviews with new candidates.
- If after any interview you decide that the new candidate is better than the current office assistant, you will fire the current assistant and hire the new candidate.
- Suppose each interview has a cost c_i and each hiring has a cost c_h.
- Then, if you make m hiring decisions in n days, your total cost will be $nc_i + mc_h$. Note that this depends on the number of hiring decisions, which depends on the order of the candidates interviewed.

Hiring Problem (cont'd)

- Best case: The best candidate arrives first => $m = 1$
- Worst case: Candidates arrive with increasing qualifications => $m = n$
- Average case analysis: Candidates arrive with random order of qualifications
  - If candidate i has a qualification rank given by $rank(i)$, then m is a function of the ordered list $\langle rank(1), rank(2), \dots, rank(n)\rangle$
  - $\langle rank(1), rank(2), \dots, rank(n)\rangle$ is a permutation of $\langle 1,2, \dots, n\rangle$
  - Random order of candidates simply means that the list of their ranks is equally likely to be one of the $n!$ possible permutations
- To ensure random input, the algorithm can start by calculating a random permutation of the input list

Example:

Assume candidates arrive with random order of qualifications

- What is the probability of the best case (i.e. hiring only once)?

  Best candidate should come first => $\mathrm{P}(m=1)=1/n$

- What is the probability of the worst case (i.e. hiring *n* times)?

  Strictly increasing qualifications => $\mathrm{P}(m=n)=1/n!$

- What is the probability of hiring twice?

  Let the best candidate arrive k^{th}. We must have $k > 1$

  There must be only one hire (the first candidate) before k.

$$\mathrm{P}(m=2)=\sum_{k=2}^{n}\frac{1}{n}P(best\ in\ 1..k-1\ is\ at\ 1)=\frac{1}{n}\sum_{k=2}^{n}\frac{1}{k-1}=\frac{1}{n}\sum_{k=1}^{n-1}\frac{1}{k}$$

Indicator Random Variables

- They provide a convenient method to represent the occurrence of events
- If A is an event in the sample space, then the indicator function associated with A is defined as:

$$I\{A\} = \begin{cases} 1 & \text{if } A \text{ occurs} \\ 0 & \text{otherwise} \end{cases}$$

- Example: For a coin toss, if H is the event that the coin is heads, then we can define an indicator random variable for H as:

$$X_H = I\{H\} = \begin{cases} 1 & \text{if heads} \\ 0 & \text{if tails} \end{cases}$$

- The expected value of an indicator random variable is equal to the probability of the associated event:

$$E\{X_H\} = 1 \cdot \Pr\{H\} + 0 \cdot \Pr\{\overline{H}\} = \Pr\{H\}$$

Analysis of Hiring Problem using Indicator Random Variables

- The expected cost depends on the expected number of times we hire a new office assistant.
- One way to calculate this expected value is defining X as the (random) number of times we hire. Then,

$$E[X] = \sum_{x=1}^{n} x \Pr\{X = x\}$$

- However, this requires calculation of $\Pr\{X = x\}$.
- Instead, define $X_i = I\{\text{candidate } i \text{ is hired}\}$.
- Then, $X = X_1 + X_2 + \cdots + X_n$, and

$$E[X] = \sum_{i=1}^{n} E[X_i] = \sum_{i=1}^{n} \Pr\{\text{candidate } i \text{ is hired}\}$$

Analysis of Hiring Problem using Indicator Random Variables (cont'd)

- The candidate i is hired if and only if i is the best candidate among the candidates $1,2,\ldots,i$.
- If candidates are interviewed in random order of qualification (as we assumed for the average analysis), then

$$\Pr\{\text{candidate } i \text{ is hired}\} = 1/i$$

- Therefore,

$$E[X] = \sum_{i=1}^{n} 1/i = \ln n + O(1)$$

- Thus, if we interview n candidates, then the average hiring cost will be $c_h \ln n$. Since interview cost is $c_i n$, even if $c_h > c_i$, the interview cost will dominate for large n.

Randomized Algorithms

- We have seen that if the input to hiring problem is random, we can find the average cost of the algorithm.
- If the input is not random, then we can still randomize it inside the algorithm and use the same result.
- For the hiring problem, compute a random permutation of the n candidates to determine their interview order.

Random Permutations by Sorting

- One way to calculate a random permutation of a given array A is first to assign a random priority $P(i)$ to each array element $A(i)$.
- Then A is sorted according to these priorities.

  Example: $A = \langle 1,2,3,4 \rangle$ and suppose the random priorities are $P = \langle 36,3,62,19 \rangle$

  Then, the permuted array will be $B = \langle 2,4,1,3 \rangle$
- Algorithm:

Permute-By-Sorting(A)

n = $A.length$

for i = 1 to n

  $P[i]$ = random$(1, n^3)$

sort A using P as sort keys

Random Permutations by Randomizing in Place

- Another algorithm to calculate a random permutation of a given array A.
- Operates in $O(n)$ time.
- At each iteration, it swaps an element with a "future" element and moves on to the next element.
- Algorithm:

Randomize-In-Place(A)

$n = A.length$

for $i = 1$ to n
    $j = \text{random}(i, n)$
    $\text{tmp} = A[i]$
    $A[i] = A[j]$
    $A[j] = \text{tmp}$

Example:

Do these versions produce random permutations?

Permute-Future(A)

$n = A.length$

for $i = 1$ to $n - 1$

swap $A[i]$ with $A[\text{random}(i+1, n)]$

No. The first element cannot remain at A[1], the second element cannot remain at A[2], … etc

Permute-All(A)

$n = A.length$

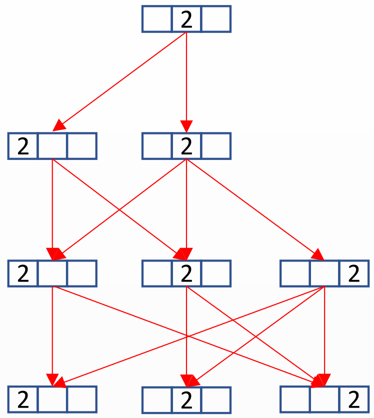
for $i = 1$ to n

swap $A[i]$ with $A[\text{random}(1, n)]$

No. To see this, let A=[1,2,3] (n=3)

and calculate the probability of where 2 will end up -> (next slide)

Example (cont'd):



Permute-All(A)

$n = A.length$

for $i = 1$ to n

  swap $A[i]$ with $A[\text{random}(1, n)]$

After first iteration: $P(2 \text{ is at } A[1]) = 1/3$ $P(2 \text{ is at } A[2]) = 2/3$

After second iteration: $P(2 \text{ is at } A[1]) = (1/3)(2/3)+(2/3)(1/3) = 4/9$

$P(2 \text{ is at } A[2]) = (1/3)(1/3)+(2/3)(1/3) = 1/3$

$P(2 \text{ is at } A[3]) = (2/3)(1/3) = 2/9$

After third iteration: $P(2 \text{ is at } A[1]) = (4/9)(2/3)+(2/9)(1/3) = 10/27$

$P(2 \text{ is at } A[2]) = (1/3)(2/3)+(2/9)(1/3) = 8/27$

$P(2 \text{ is at } A[3]) = (4/9)(1/3)+(1/3)(1/3)+(2/9)(1/3) = 1/3$

Not equal

Example: Random Subset

Suppose we want to create a random sample of the set {1,2,3,…,n}, that is, an m-element subset S, where 0≤m≤n, such that each m-subset is equally likely to be created. One way would be:

Rand-Subset(m, n)

for i = 1 to n

  $A[i] = i$

Randomize-In-Place(A)

return $A[1..m]$

For each $k = 1,2, \dots, n$,

$$P(k \in S) = P(k \in A[1..m]) = m/n$$

Each number is equally likely to appear in the output; therefore it is indeed selected uniformly.

- This method would make n calls to the RANDOM procedure.
- If n is much larger than m, we can create a random subset with fewer calls to RANDOM (Exercise 5.3-7 – HW)

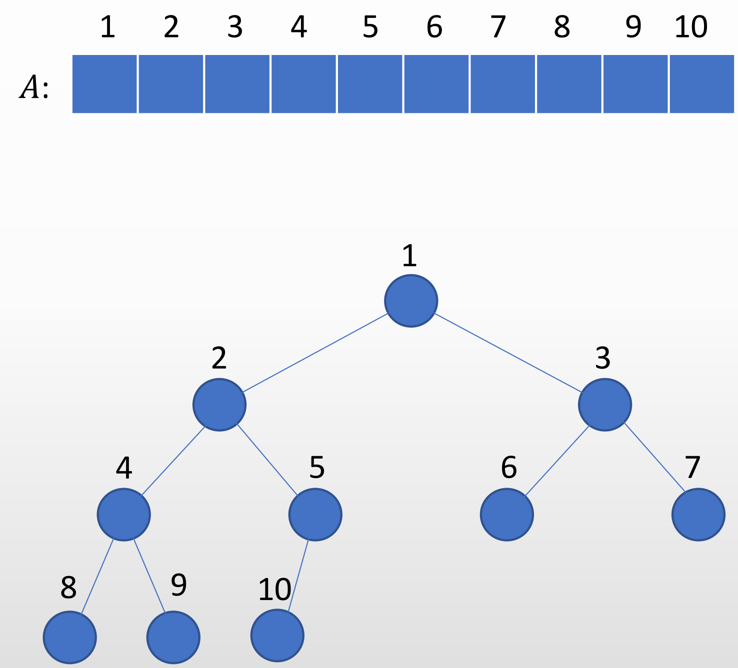
Sorting Algorithms - Heapsort

Running Times for Sorting Algorithms

Algorithm	Worst-case Running Time	Average-case Running Time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n\lg n)$	$\Theta(n\lg n)$
Heapsort	$O(n\lg n)$	$O(n\lg n)$
Quicksort	$\Theta(n^2)$	$\Theta(n\lg n)$
Counting sort	$\Theta(k+n)$	$\Theta(k+n)$
Radix sort	$\Theta(d(k+n))$	$\Theta(d(k+n))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$

Heaps

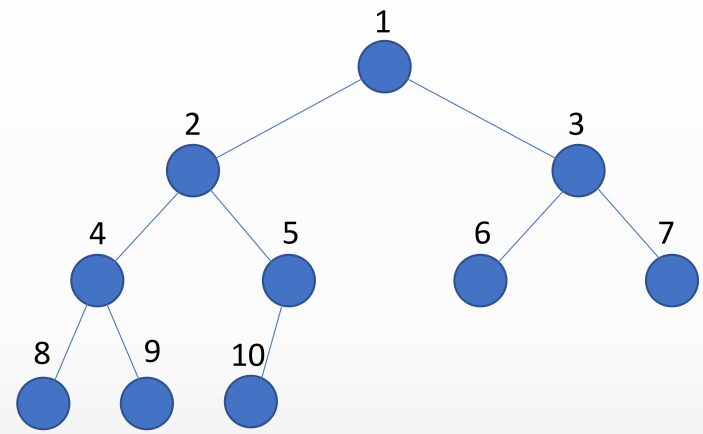
- A (binary) heap data structure is a nearly-complete binary tree, where each node corresponds to an element in the array.
- All levels of the tree is completely filled, except possibly the lowest – the lowest is filled from left up to a point.
- An array A that represents a heap should have *A.heap-size* attribute that shows how many elements in A are in the heap. This can be different than the length of the array $A.length$.
- The root of the tree is $A[1]$.



Heaps (cont'd)

- Given the index i of a node,
  Left(i) =$2i = (i \ll 1)$
  Right(i) =$2i + 1 = (i \ll 1) + 1$
  Parent(i) = $\lfloor i/2 \rfloor = i \gg 1$
- The <u>height</u> of a node is the number of edges on the longest path from the node to a leaf.
- The height of the heap is the height of its root, $\Theta(\lg n)$ if heap has n nodes.
- A <u>max-heap</u> is a heap where the value stored in any node is not larger than the value at its parent, i.e.

$$A[i] \le A[\text{Parent}(i)] \text{ for all } i$$



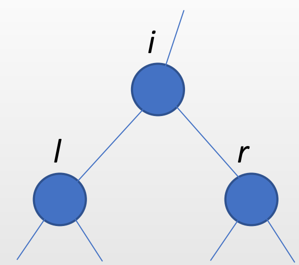
Maintaining Heap Property

- Check if the given element satisfies max-heap property; push it down to correct level if it doesn't
- Inputs: Heap array A and an index i
- Assumes that the trees rooted at Left(i) and Right(i) already satisfy the max-heap property

Max-Heapify(*A*,*i*)

l = Left(*i*)
r = Right(*i*)
if *l* <= *A.heap-size* and *A*[*l*] > *A*[*i*]
    largest = *l*
else *largest* = *i*
if *r* <= *A.heap-size* and *A*[*r*] > *A*[*largest*]
    largest = *r*
if *largest* <> *i*
    swap *A*[*i*] with *A*[*largest*]
    Max-Heapify(*A*, *largest*)

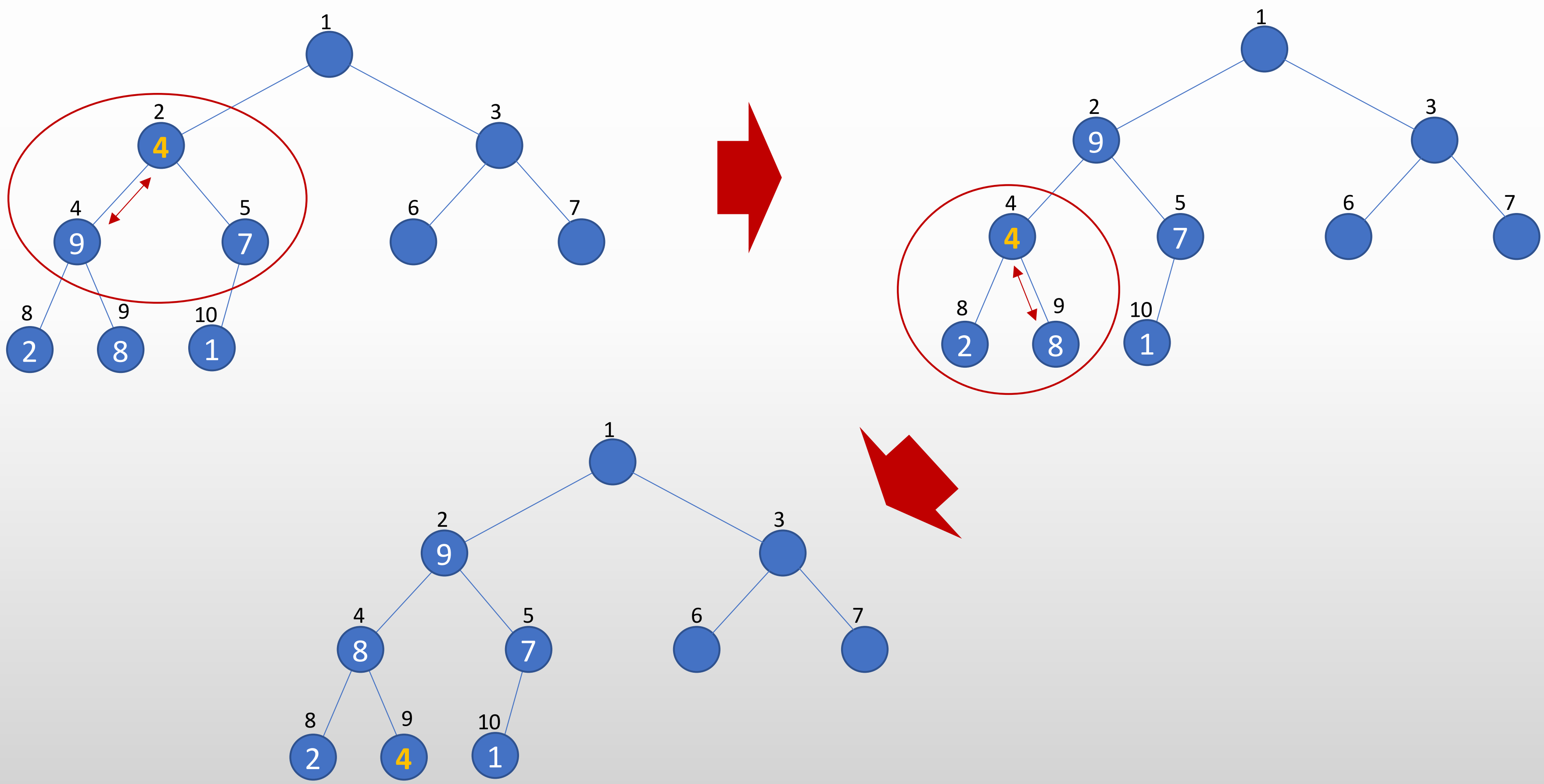
Find the largest of $A(i), A(l), A(r)$



If the largest is $A(i)$ then done

Otherwise swap $A(i)$ with the largest and repeat from that node

Example



1
2
3
4
5
6
7
8
9
10
4
9
7
2
8
1
1
2
3
4
5
6
7
8
9
10
9
4
7
2
8
1
1
2
3
4
5
6
7
8
9
10
9
8
7
2
4
1

Complexity of Max-Heapify

```
Max-Heapify(A,i)
l = Left(i)
r = Right(i)
if l <= A.heap-size and A[l] > A[i]
    largest = l
else largest = i
if r <= A.heap-size and A[r] > A[largest]
    largest = r
if largest <> i
    swap A[i] with A[largest]
    Max-Heapify(A, largest)
```

Constant time, $\Theta(1)$

In the worst case the subtree can have $2n/3$ nodes

Recurrence: $T(n) \leq T\left(\frac{2n}{3}\right) + \Theta(1)$

By the master theorem, this has solution $T(n) = O(\lg n)$

Building Heaps

- Given an array $A[1..n]$, where $n = A.length$, how do you build a heap?
- Observation: The indices that correspond to heap leaves are already “sub-heaps” since they don’t have any child nodes.
- Starting from the leaves, we can move upwards towards the root, combining the “sub-heaps” to form larger and larger heaps, until we have the entire array.
  - i.e. use Max-Heapify starting from the bottom towards up
- To start, which indices correspond to leaves?
  - Leaves cannot have children, so if i is a leaf, then $2i$ should not be valid index

$$2\left\lfloor\frac{n}{2}\right\rfloor \leq n \text{ (valid), but } 2\left(\left\lfloor\frac{n}{2}\right\rfloor + 1\right) > n \text{ (invalid)}$$

  - Therefore the leaves correspond to indices $\lfloor n/2 \rfloor + 1, \ldots, n$

Build-Max-Heap(A)

A.heap-size = *A.length*
for i = $\lfloor A.length/2 \rfloor$ downto 1
    Max-Heapify(A, i)

Building Heaps (cont'd)

Build-Max-Heap(A)

A.heap-size = *A.length*

for i = $\lfloor A.length/2 \rfloor$ downto 1

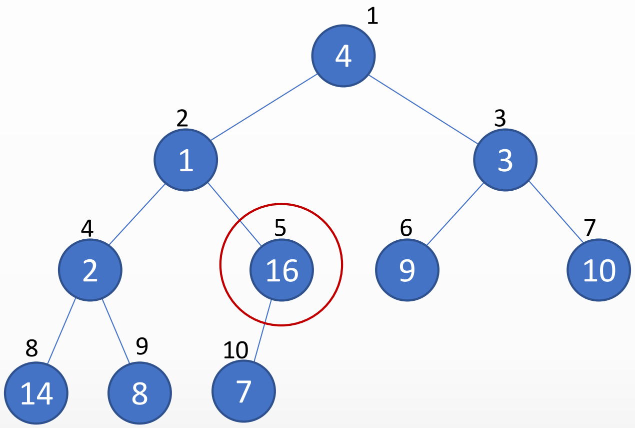
Max-Heapify(A, i)

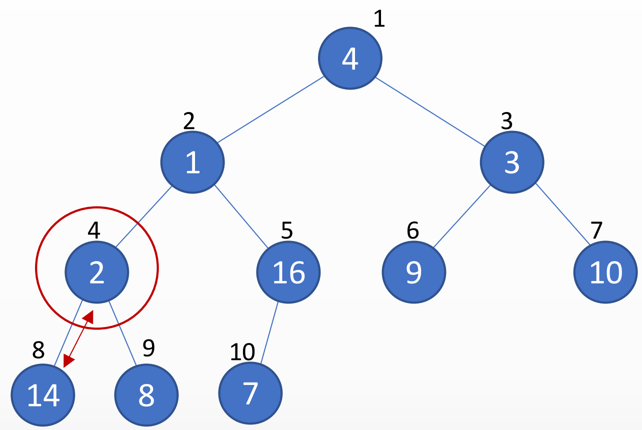
Correctness of the Algorithm:

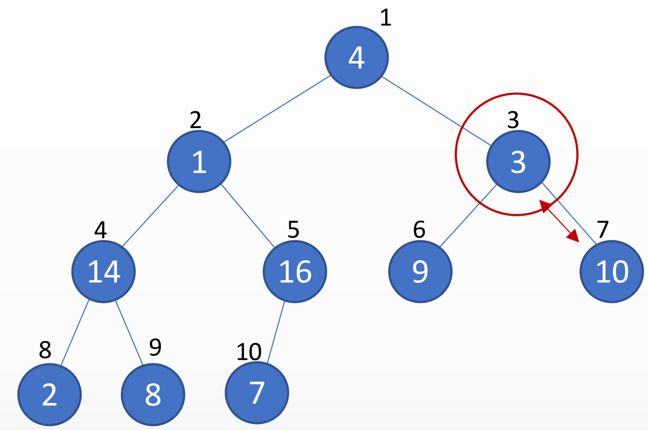
- Loop invariant: Each of the nodes $i+1, \dots, n$ is the root of a max-heap at the start of each iteration of the loop
- Initialization:
  For $i = \lfloor n/2 \rfloor$, the nodes $\lfloor n/2 \rfloor + 1, \dots, n$ are all leaves
- Maintenance:
  The loop calls Max-Heapify with the index i
  The children of node i are both higher indexed, and therefore they are already roots of two max-heaps
  Max-Heapify makes node i the root of a max-heap, so the invariant holds true when i is reduced at the next iteration
- Termination:
  When the algorithms terminates after $i = 1$ is processed, the entire array is a max-heap with the root at node 1.

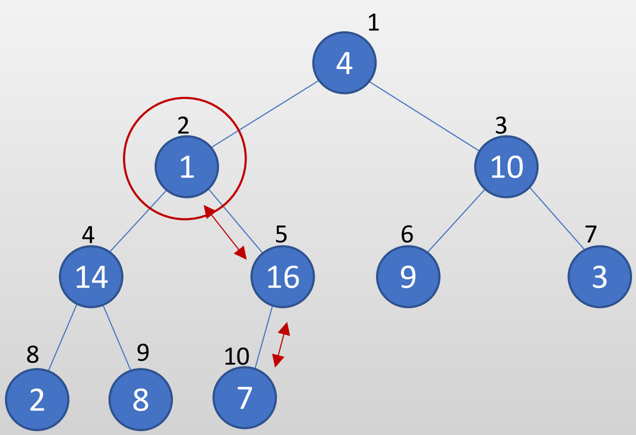
Example

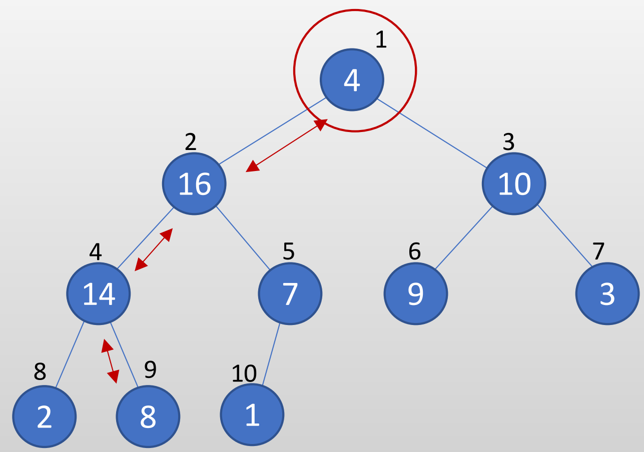
$A = \langle 4,1,3,2,16,9,10,14,8,7 \rangle$



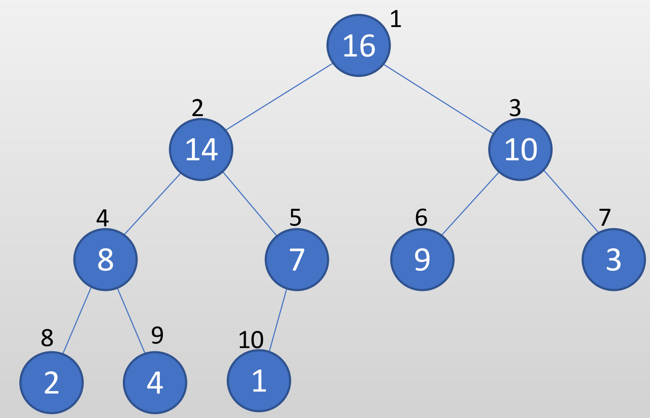








$A' = \langle 16,14,10,8,7,9,3,2,4,1 \rangle$



Running Time of Build-Max-Heap

- Simple upper bound:
  - Max-Heapify runs at $O(\lg n)$ time and we call it $O(n)$ times, therefore running time of Build-Max-Heap if $O(n \lg n)$
  - Correct upper bound, but not asymptotically tight
- A tighter bound:
  - If Max-Heapify is called for a node with height h, then it costs $O(h)$
  - There are at most $\lceil n/2^{h+1} \rceil$ nodes of height h
  - So, a tighter bound on running time is:

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

Build-Max-Heap(A)

A.heap-size = *A.length*

for i = $\lfloor A.length/2 \rfloor$ downto 1
  Max-Heapify(A, i)

Heaps can be built in linear time

Heapsort Algorithm

- Same sorting problem as before – we want to sort the elements of an array A
- Idea:

  If we build a max-heap from A, we know its largest element will be at $A[1]$.

  We can now remove $A[1]$ from the heap, and repeat with the remaining elements to find the second largest element, … etc

Heapsort(A)

Build-Max-Heap(A)

for i = $A.length$ downto 2

  swap $A[1]$ with $A[i]$

  $A.heap\text{-}size = A.heap\text{-}size - 1$

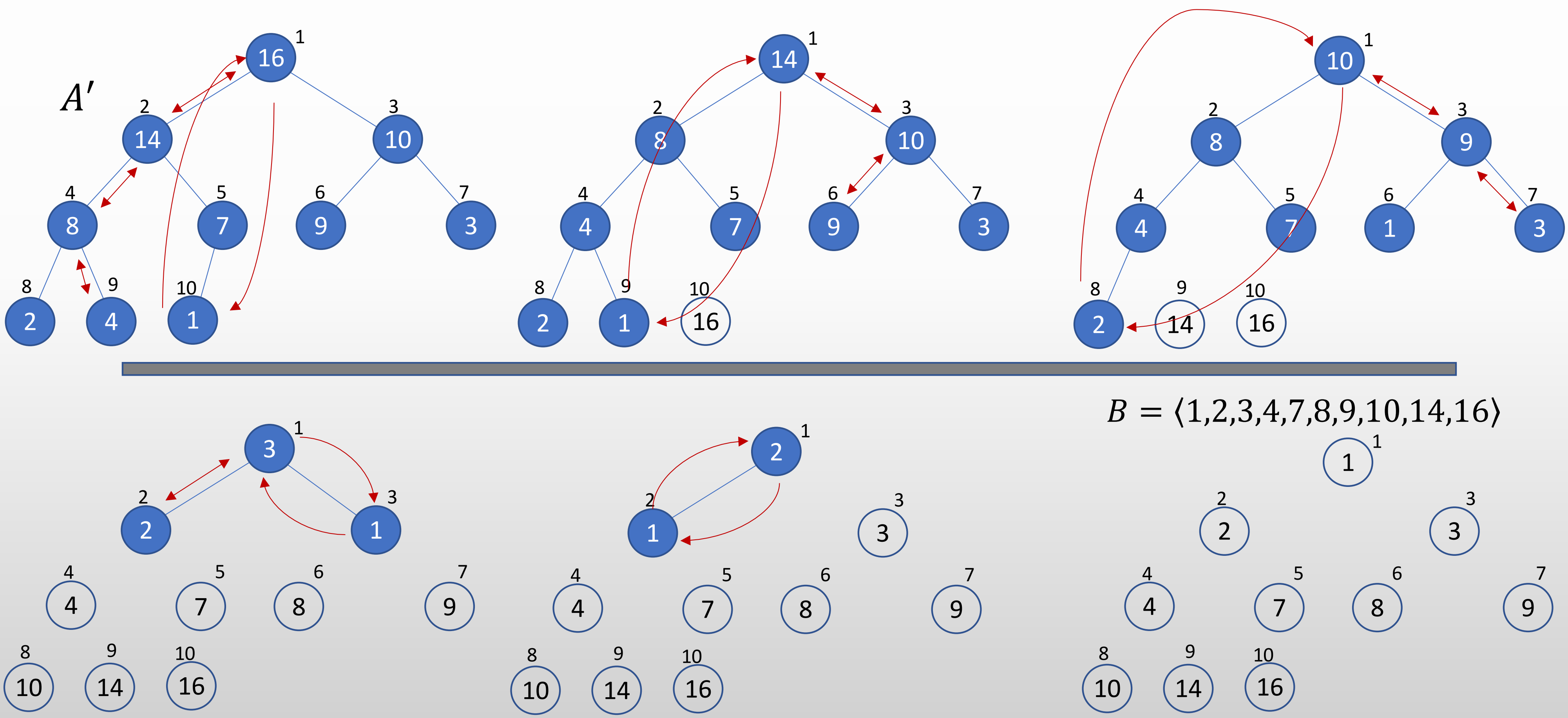
  Max-Heapify(A, 1)

Moves the largest element to the end, which is the correct location in the sorted array; and reduces heap size so it is not touched again

Only the element swapped with $A[1]$ can be out of place; the rest are already max-heaps

Example

$A = \langle 4,1,3,2,16,9,10,14,8,7 \rangle$ $\xrightarrow{\text{Build-Max-Heap}}$ $A' = \langle 16,14,10,8,7,9,3,2,4,1 \rangle$



$B = \langle 1,2,3,4,7,8,9,10,14,16 \rangle$

Running Time of Heapsort

Heapsort(A)

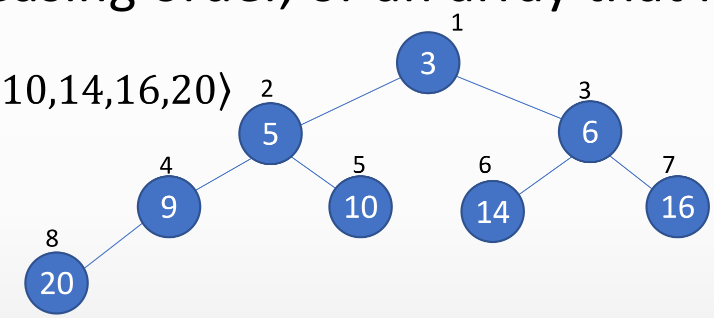
```
Build-Max-Heap(A)
for i = A.length downto 2
    swap A[1] with A[i]
    A.heap-size = A.heap-size – 1
    Max-Heapify(A, 1)
```

- Calling Build-Max-Heap takes $O(n)$
- The for loop executes $n - 1$ times, and each iteration calls Max-Heapify with the entire tree, i.e. each iteration takes $O(\lg n)$ time
- Therefore, the for loop dominates the complexity of heapsort, which requires $O(n \lg n)$ time

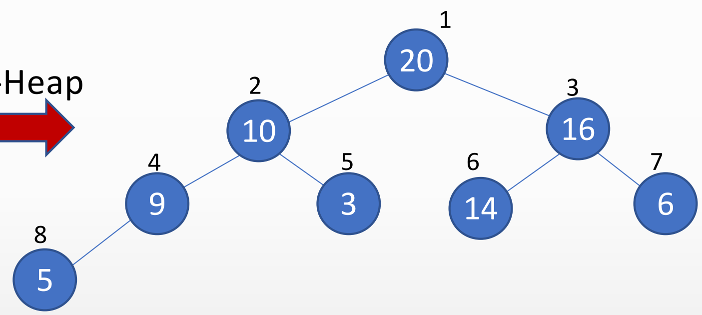
Example:

- Which one would HEAPSORT sort quicker: an array that is already sorted in increasing order, or an array that is sorted in decreasing order?

$A = \langle 3,5,6,9,10,14,16,20 \rangle$

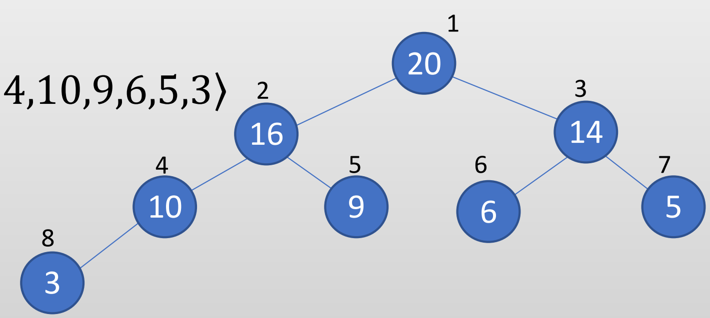


Build-Max-Heap



(takes longer to process)

$A = \langle 20,16,14,10,9,6,5,3 \rangle$



(already arranged as Max Heaps)

Priority Queues

- A priority queue is a data structure that maintains the elements of a set S.
- Each element of S has a special value called the key.
- A (max) priority queue supports the following operations:
  - Insert(S, x) , to insert a new element x into S
  - Maximum(S), to return the element in S with the largest key
  - Extract-Max(S), to remove and return the element in S with the largest key
  - Increase-Key(S, x, k), to increase element x's key to the new value k

    (k should be larger than or equal to x's current key value)

- Sample applications: Scheduling tasks on a computer, servicing troubleshooting tickets, etc

Using Max-Heaps for Priority Queues

Each of the four operations can be implemented using heaps

Heap-Maximum(A)

return $A[1]$

Heap-Extract-Max(A)

$max = A[1]$

$A[1] = A[A.heap\text{-}size]$

$A.heap\text{-}size = A.heap\text{-}size - 1$

Max-Heapify(A,1)

return max

Heap-Increase-Key(A,i,key)

$A[i] = key$

```
while i >1 and A[Parent(i)] < A[i]
    swap A[i] with A[Parent(i)]
    i = Parent(i)
```

Heap-Insert(A,key)

$A.heap\text{-}size = A.heap\text{-}size + 1$

$A[A.heap\text{-}size] = -\infty$

Heap-Increase-Key(A, $A.heap\text{-}size$, key)