CSE 548: Analysis of Algorithms

Prerequisites Review 4
(Heaps and Heapsort)

Rezaul A. Chowdhury

Department of Computer Science
SUNY Stony Brook
Fall 2019

Selection Sort

Input: An array $A[\,1 : n\,]$ of n numbers.

Output: Elements of $A[\,1 : n\,]$ rearranged in non-decreasing order of value.

SELECTION-SORT (A)

```
1.  for j = A.length downto 2
2.      // find the index of an entry with the largest value in A[1..j]
3.      max = 1
4.      for i = 2 to j
5.          if A[i] > A[max]
6.              max = i
7.      // swap A[j] and A[max]
8.      A[j] ↔ A[max]
```

This way of finding the index of an entry with the largest value in a subarray of length m takes $\Theta(m)$ time, which is bad!

Selection Sort

```
SELECTION-SORT ( A )
1.  for j = A.length downto 2
2.      // find the index of an entry with the largest value in A[1..j]
3.      max = 1
4.      for i = 2 to j
5.          if A[i] > A[max]
6.              max = i
7.      // swap A[j] and A[max]
8.      A[j] ↔ A[max]
```

This way of finding the index of an entry with the largest value in a subarray of length m takes $\Theta(m)$ time, which is bad!

Let $L(m)$ be the time needed to find the index of an entry with the largest value in a subarray of length m.

Then running time of SELECTION-SORT, $T(n) = \sum_{2 \le j \le n} L(j)$

$$= \sum_{2 \le j \le n} \Theta(j) = \Theta\left(\sum_{2 \le j \le n} j \right) = \Theta(n^2)$$

Selection Sort

```
SELECTION-SORT ( A )
1.  for j = A.length downto 2
2.      // find the index of an entry with the largest value in A[1..j]
3.      max = 1
4.      for i = 2 to j
5.          if A[i] > A[max]
6.              max = i
7.      // swap A[j] and A[max]
8.      A[j] ↔ A[max]
```

This way of finding the index of an entry with the largest value in a subarray of length m takes $\Theta(m)$ time, which is bad!

If we can decrease $L(m)$, then the running time of SELECTION-SORT will also decrease. For example, if we have $L(m) = O(\log m)$,

running time of SELECTION-SORT will be, $T(n) = \sum_{2 \le j \le n} L(j)$

$$= \sum_{2 \le j \le n} O(\log(j)) = O\left(\sum_{2 \le j \le n} \log(j) \right) = O(n \log n)$$

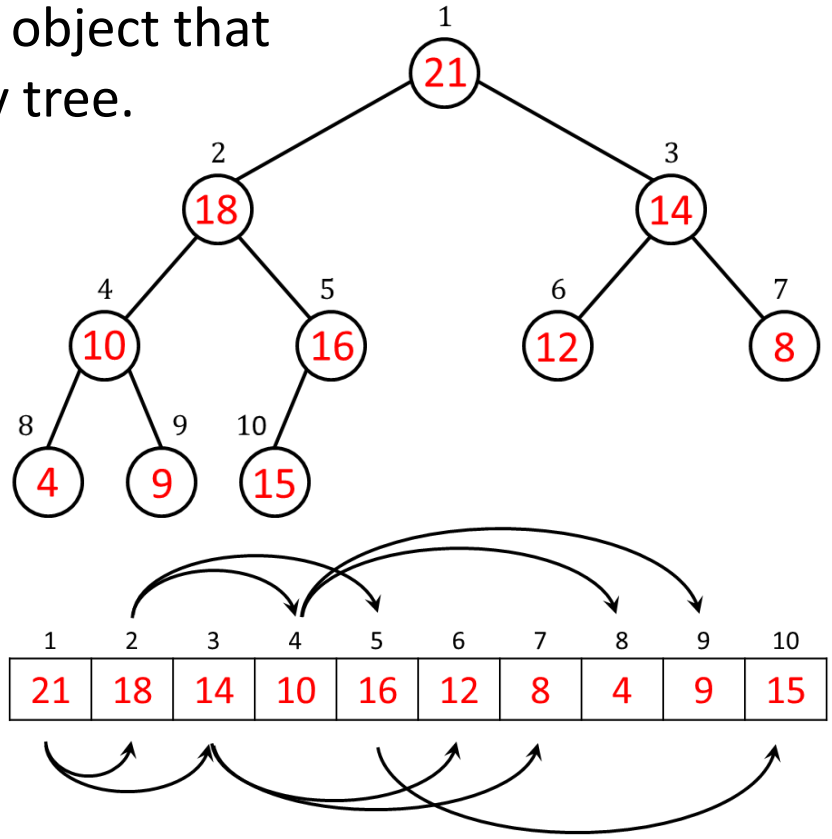
How can you decrease $L(m)$ to $O(\log m)$?

Heap (Binary Heap)

A (*binary*) *heap* data structure is an array object that can be viewed as a nearly complete binary tree.

Each node of the tree corresponds to an element of the array.

The tree is completely filled on all levels except possibly the last, which is filled from the left up to a point.



An array A that represents a heap is an object with two attributes:

$A.length$, which gives the number of elements in the array.

$A.heapsize$, which represents how many elements in the heap are stored within array A.

Though $A[1..A.length]$ may contain numbers, only $A[1..A.heapsize]$ contain valid elements of the heap, where $1 \leq A.heapsize \leq A.length$.

Parent and Children

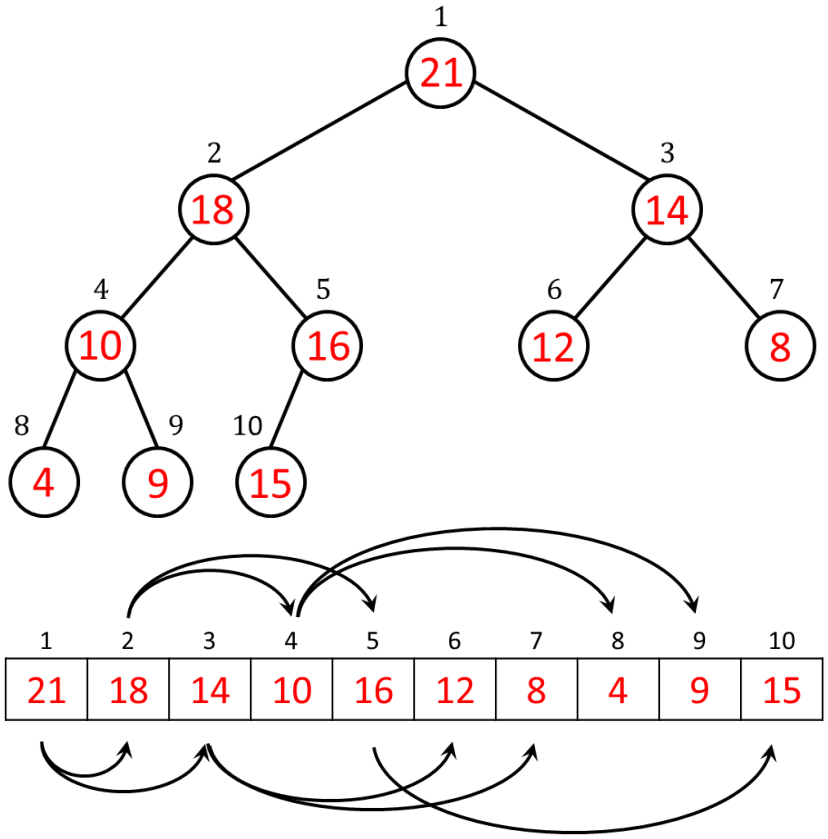
The root of the tree is $A[1]$.

A node has 0, 1 or 2 children.

A node with no child is called a *leaf*.

A node with at least one child is called an *internal node*.

Given the index i of a node, we can easily compute the indices of its *parent*, *left child* and *right child*.



PARENT (i)

1. ***return*** $\left\lfloor \frac{i}{2} \right\rfloor$

LEFT (i)

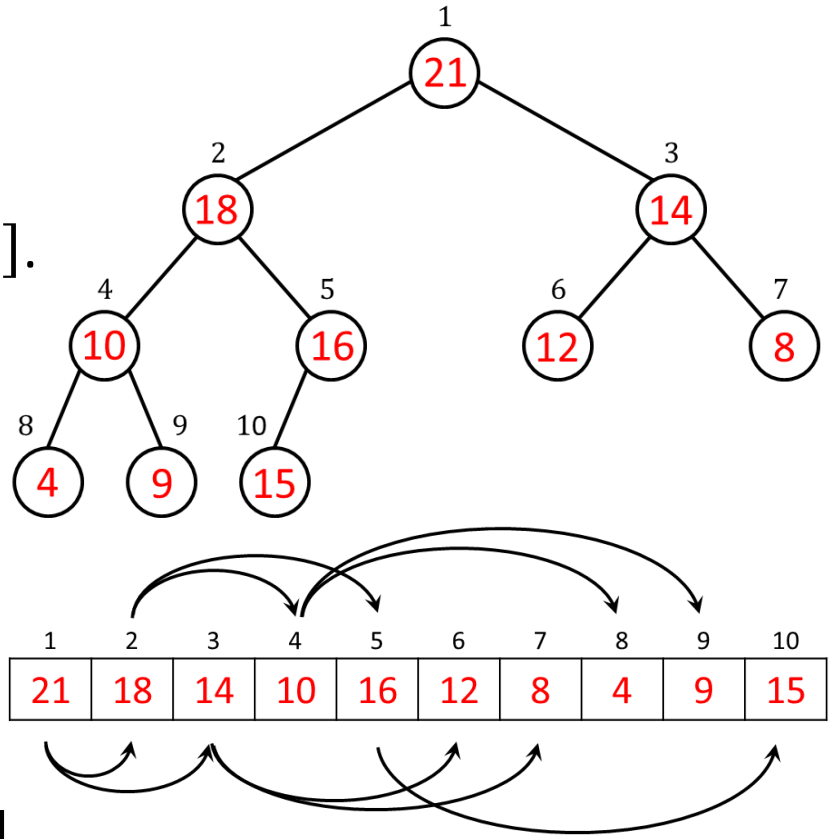
1. ***return*** $2i$

RIGHT (i)

1. ***return*** $2i + 1$

Max-Heap and Min-Heap

The root of the tree is $A[1]$.



Max-heap. Each node $i > 1$ satisfies the *max-heap property*: $A[\text{PARENT}(i)] \geq A[i]$.

Hence, the largest element in a max-heap is stored at the root.

We will use max-heaps in the *heapsort* algorithm which can be viewed as improved selection sort.

Min-heap. Each node $i > 1$ satisfies the *min-heap property*: $A[\text{PARENT}(i)] \leq A[i]$.

Hence, the smallest element in a min-heap is stored at the root.

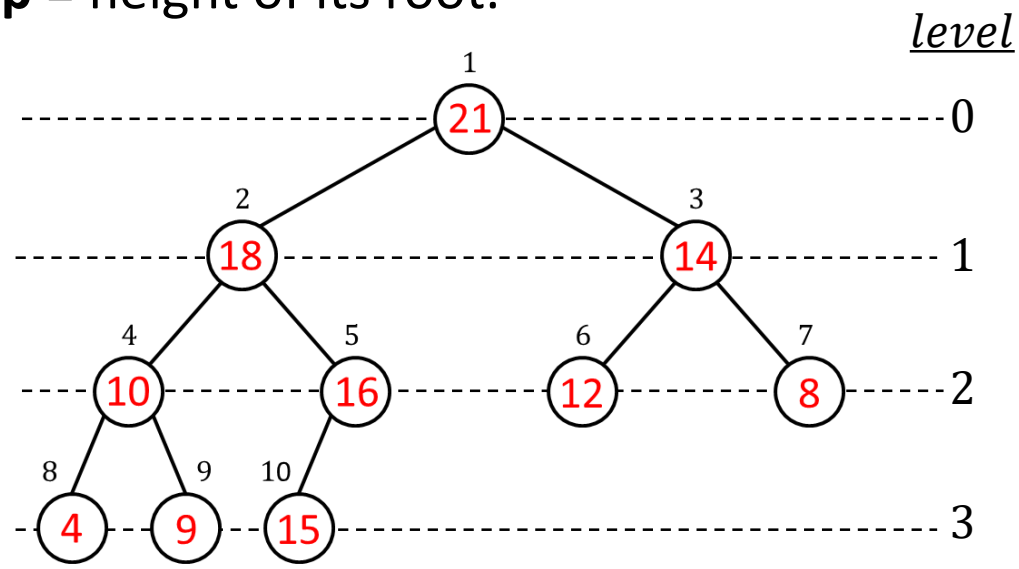
Min-heaps commonly implement priority queues which have many applications, e.g., in shortest paths computation.

Height and Levels of a Heap

The root of the tree is $A[1]$.

Height of a node = Number of edges on the longest simple downward path from that node to a leaf.

Height of a heap = height of its root.



Levels:

Level of the root, $\text{LEVEL}(1) = 0$

Level of node $i > 1$, $\text{LEVEL}(i) = \text{LEVEL}\big(\text{PARENT}(i)\big) + 1$

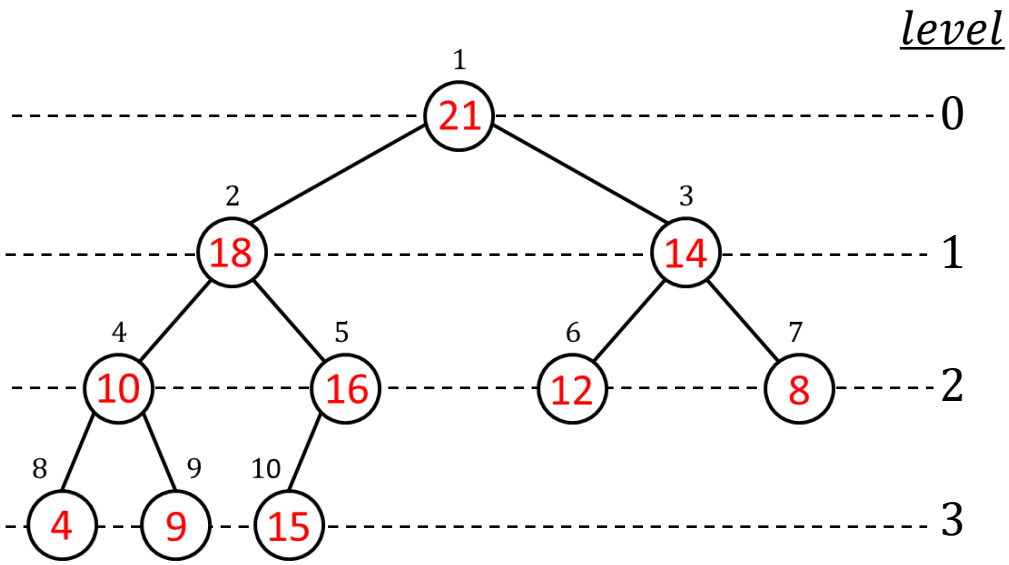
A heap of height h has exactly $h + 1$ levels, numbered from 0 to h.

Height of an n-node Binary Heap

Let h be the height of a heap containing $n > 0$ elements.

So, the heap will have exactly $h + 1$ levels.

Let n_l be the number of nodes at level l, where $0 \leq l \leq h$.



Clearly, $n_l = 2^l$ for $0 \leq l \leq h - 1$,
and $1 \leq n_l \leq 2^l$ for $l = h$.

Also $n = n_0 + n_1 + \cdots + n_h = \sum_{l=0}^{h} n_l$.

Height of an n-node Binary Heap

We have, $n = \sum_{l=0}^{h} n_l = n_h + \sum_{l=0}^{h-1} n_l = n_h + \sum_{l=0}^{h-1} 2^l = n_h + \left(2^h - 1\right)$.

But $1 \leq n_h \leq 2^h$

$\Rightarrow 1 + \left(2^h - 1\right) \leq n_h + \left(2^h - 1\right) \leq 2^h + \left(2^h - 1\right)$

$\Rightarrow 2^h \leq n \leq 2^{h+1} - 1$

$\Rightarrow 2^h \leq n < 2^{h+1}$

$\Rightarrow \log_2 n - 1 < h \leq \log_2 n$

Since h is an integer, and the only integer $> \log_2 n - 1$ and $\leq \log_2 n$ is $\lfloor \log_2 n \rfloor$, we have $h = \lfloor \log_2 n \rfloor$.

Maintaining Heap Property

Input: An array A and an index i into the array with the subtrees rooted at $\text{LEFT}(i)$ and $\text{RIGHT}(i)$ are max-heaps, but $A[i]$ might be smaller than its children and thus violating the max-heap property.

Output: Array A with its elements rearranged so that the subtree rooted at index i is a max-heap.

MAX-HEAPIFY (A, i)

1. $l = \text{LEFT}(i)$
2. $r = \text{RIGHT}(i)$
3. ***if*** $l \leq A.heapsize$ and $A[l] > A[i]$
4. $\quad largest = l$
5. ***else*** $largest = i$
6. ***if*** $r \leq A.heapsize$ and $A[r] > A[largest]$
7. $\quad largest = r$
8. ***if*** $largest \neq i$
9. $\quad$ exchange $A[i]$ with $A[largest]$
10. $\quad$ MAX-HEAPIFY (A, $largest$)

Building a Max-Heap

Input: An array $A[1:n]$, where $n = A.length$.

Output: Array A with its elements rearranged so that the entire array is now a max-heap.

```
BUILD-MAX-HEAP ( A )
1.   A.heapsize = A.length
2.   for i = ⌊A.length/2⌋ downto 1
3.       MAX-HEAPIFY ( A, i )
```

The Heapsort Algorithm

Input: An array $A[\,1 : n\,]$ of n numbers.

Output: Elements of $A[\,1 : n\,]$ rearranged in non-decreasing order of value.

HEAPSORT (A)

1. BUILD-MAX-HEAP (A)
2. ***for*** $i = A.length$ ***downto*** 2
3. exchange $A[1]$ with $A[i]$
4. $A.heapsize = A.heapsize - 1$
5. MAX-HEAPIFY (A, 1)

Priority Queues

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

A *max-priority queue* supports the following operations:

$\textsc{Insert}(S, x)$ inserts the element x into the set S, which is equivalent to the operation $S = S \cup \{x\}$.

$\textsc{Maximum}(S)$ returns the element of S with the largest key.

$\textsc{Extract-Max}(S)$ removes and returns the element of S with the largest key.

$\textsc{Increase-Key}(S, x, k)$ increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value.

A Max-Heap as a Max-Priority Queue

```
HEAP-MAXIMUM ( A )
1.   return A[1]
```

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

A Max-Heap as a Max-Priority Queue

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

1. ***if*** $key < A[i]$
2. ***error*** "new key is smaller than current key"
3. $A[i] = key$
4. ***while*** $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
5. exchange $A[i]$ with $A[\text{PARENT}(i)]$
6. $i = \text{PARENT}(i)$

MAX-HEAP-INSERT (A, key)

1. $A.heapsize = A.heapsize + 1$
2. $A[A.heapsize] = -\infty$
3. HEAP-INCREASE-KEY (A, $A.heapsize$, key)