

Lecture 10: Priority Queues

01204212 Abstract Data Types and Problem Solving

Department of Computer Engineering
Faculty of Engineering, Kasetsart University
Bangkok, Thailand.



Department of
Computer Engineering
Kasetsart University



Outline

- Priority Queues
- Binary Heaps
- Application:
 - Huffman code

Priority Queues

- With an **abstract queue**, objects are linearly ordered associated by **first in, first out**
- Unlike a normal queue, an **abstract priority queue** holds a collection of objects that each will be dequeued following a **priority order**
 - i.e., each object is associated with a **priority**, and the object having the **highest priority** will be **dequeued first**
- With each enqueued object, we will associate a **nonnegative integer** (0, 1, 2, ...) where:
 - The value 0 has the highest priority, and
 - The higher the number, the lower the priority

Operations

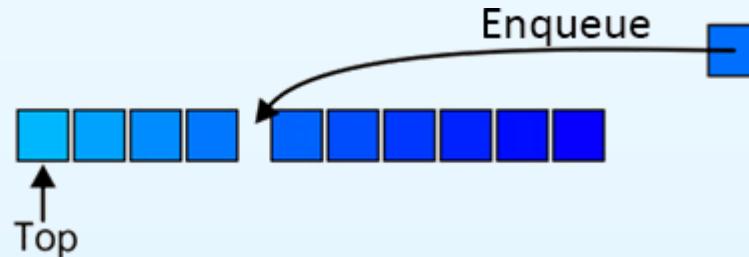
The **top** of a priority queue is the object with highest priority



Dequeueing from a priority queue removes the current highest priority object



Enqueuing places a new object into the appropriate place

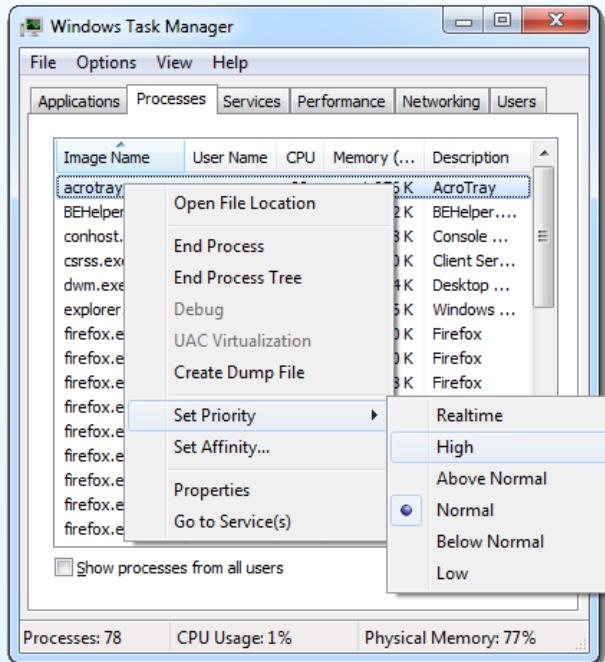


Lexicographical Priority

- Priority may also depend on **multiple variables**:
 - Two values specify a priority: (a, b)
 - A pair (a, b) has higher priority than (c, d) if
 - $a < c$, or
 - $a = c$ and $b < d$
- For example,
 - $(5,19)$, $(13,1)$, $(13,24)$, and $(15,0)$ all have higher priority than $(15,7)$

Applications

- Operating system needs to schedule jobs according to priority
 - Unix: `$ nice +15 ./a.out`
 - Reduces the priority of the execution of the routine `a.out` by 15
 - Windows: set in the Windows task Manager



Applications

- Doctors in emergency room take patients according to severity of injuries
- In traffic light, depending upon the traffic, the colors will be given priority
- Special service for members
 - e.g., airline passengers
- Huffman codes for data compression

Design for Implementation

- In addition to the **data values**, we may need to store the **priorities** (i.e., **keys**) of each object
 - Remember, the smaller the number, the higher the priority
- Therefore, we want an ADT that can efficiently perform:
 - `find_min()`
 - `delete_min()`
 - `insert()`

Which ADT?

ADT Candidates

ADT	<code>find_min()</code>	<code>delete_min()</code>	<code>insert()</code>
1. Sorted List 	$O(1)$	$O(1)$	$O(n)$
2. Unsorted List 	$O(n)$	$O(n)$	$O(1)$
3. Multiple Queues 	$O(M)$ Restrict the range of priority and require memory $\Theta(M + n)$	$O(M)$	$O(1)$
4. BST/AVL Tree 	$O(\log n)$ Look good but ... are efficient for all finds, not just <code>find_min()</code>	$O(\log n)$	$O(\log n)$

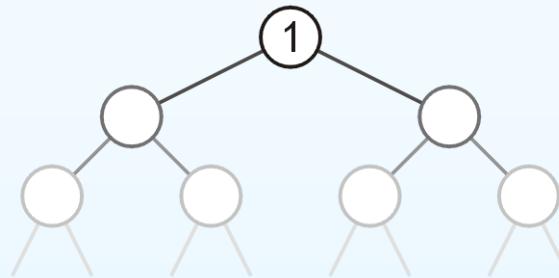
Better than a BST !!!

Can we **do better** with very limited requirements?

- Only need `find_min()`, `delete_min()`, and `insert()`
- Reduce the running time of some operations down to $\Theta(1)$

We will look at another ADT, called **heap**

- A tree with the highest priority object at the root
- We will focus on **binary heaps**
- Numerous other heaps exists:
 - d -ary heaps
 - Leftist heaps
 - Skew heaps
 - Binomial heaps
 - Fibonacci heaps
 - Bi-parental heaps



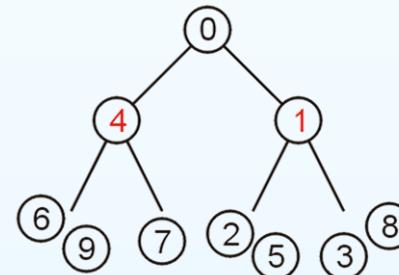
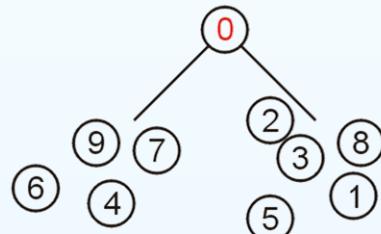
Outline

- Priority Queues
- Binary Heaps
- Application:
 - Huffman code

Definition: Binary Min Heaps

A non-empty binary tree is a **min-heap** if

- The key of the root is **less than** or **equal to** the keys associated with its left and right subtrees (if any)
- Both subtrees (if any) are also binary min-heaps

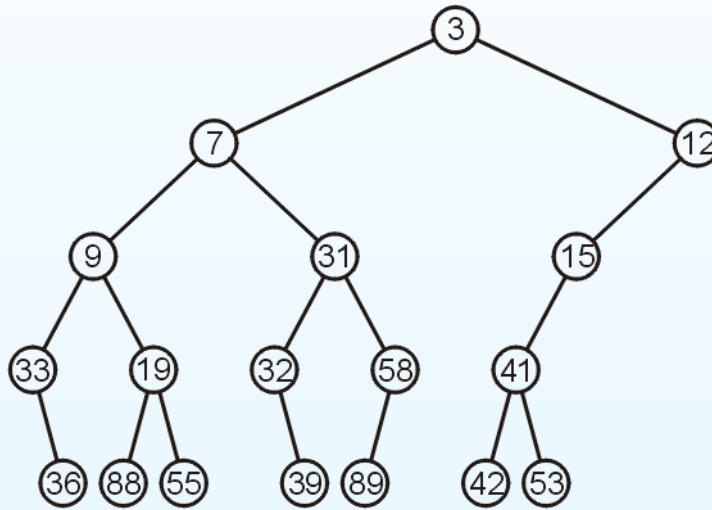


From the definition:

- A single node is a min-heap
- All keys in either subtree are greater than the key of root

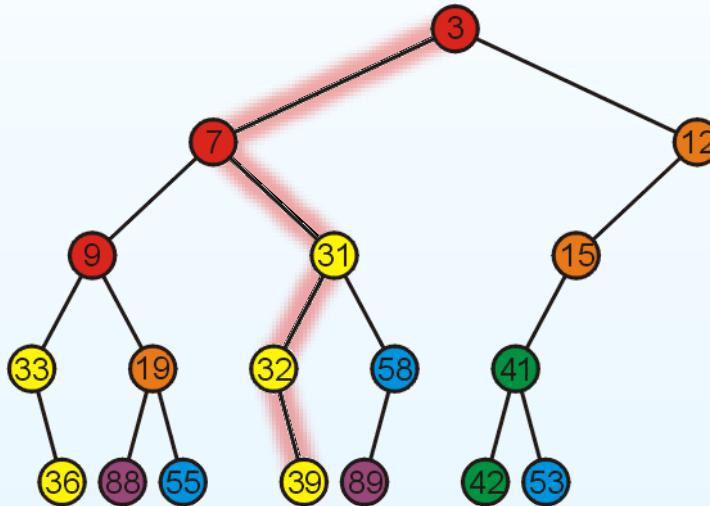
Example

A binary min-heap



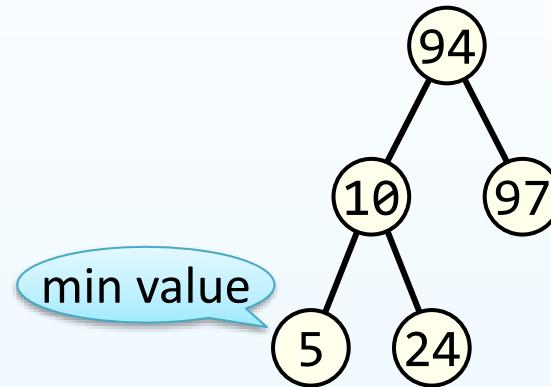
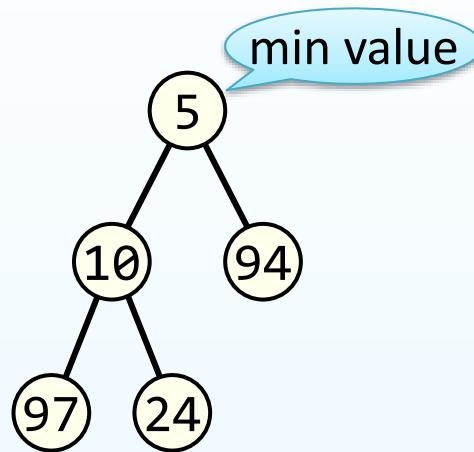
Heap Order Properties

- Each path is sorted
 - e.g., $\langle 3, 7, 31, 32, 39 \rangle$
- There is **no relationship** between the elements in the two subtrees



Binary Heap vs. Binary Search Tree

A binary heap **is NOT** a binary search tree



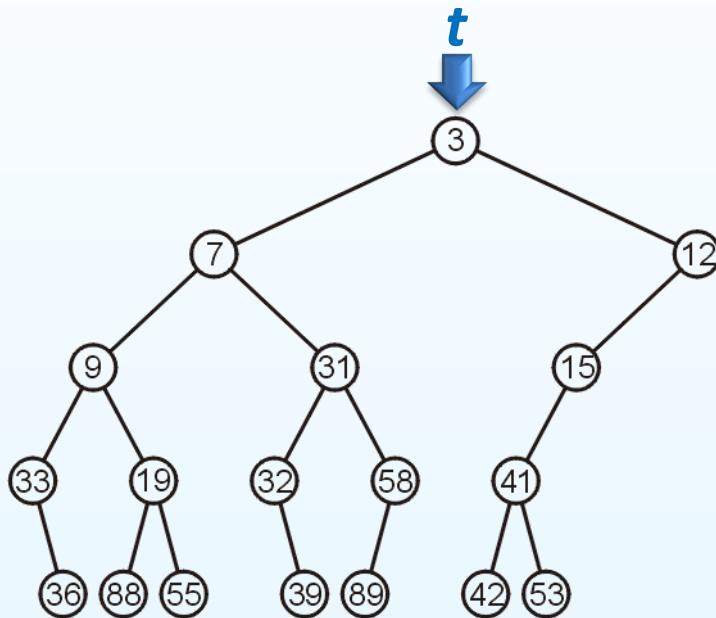
Binary min-heap: parent is less than both left and right children

Binary search tree: parent is greater than left child, but less than right child

The `find_min()` Operation

Return the minimum value in heap t

`find_min(t)`

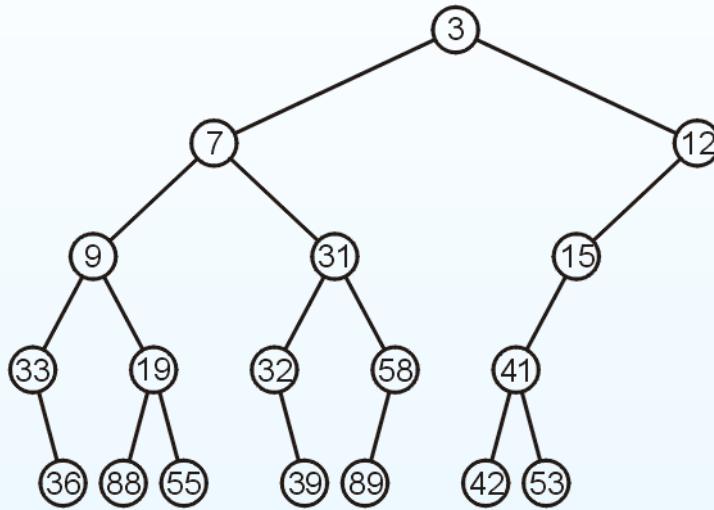


Running time:

We can easily return the value of the root in $\Theta(1)$

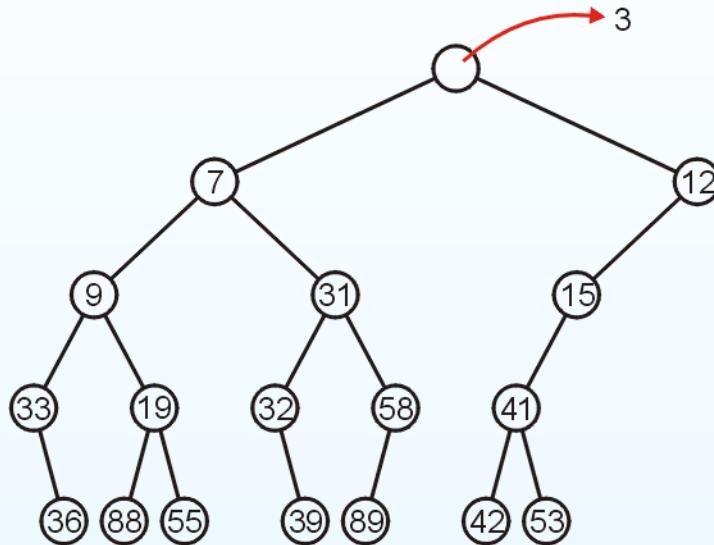
Deletion

We want to remove the minimum value from the heap:



Deletion

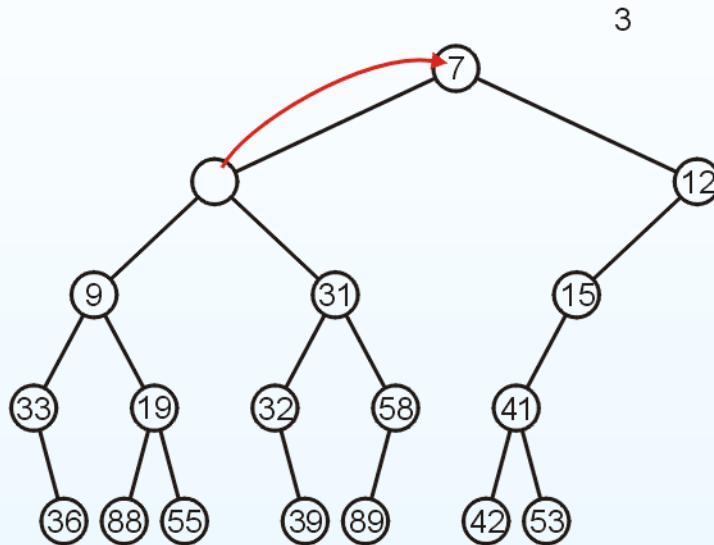
We want to remove the minimum value from the heap:



1. Remove 3

Deletion

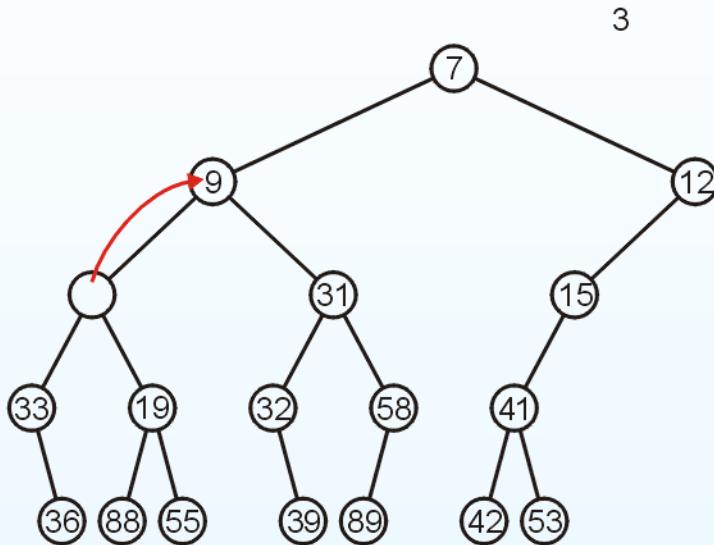
We want to remove the minimum value from the heap:



1. Remove 3
2. Promote 7 (the minimum of 7 and 12) to the root

Deletion

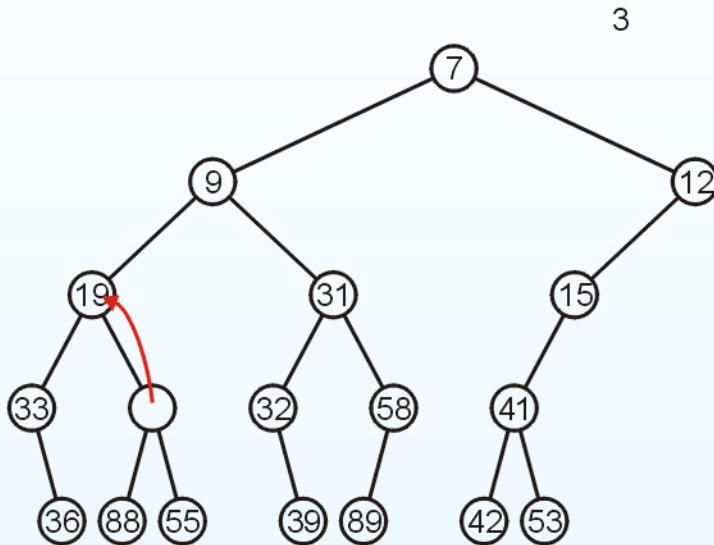
We want to remove the minimum value from the heap:



1. Remove 3
2. Promote 7 (the minimum of 7 and 12) to the root
3. Recursively, promote 9 (the minimum of 9 and 31)

Deletion

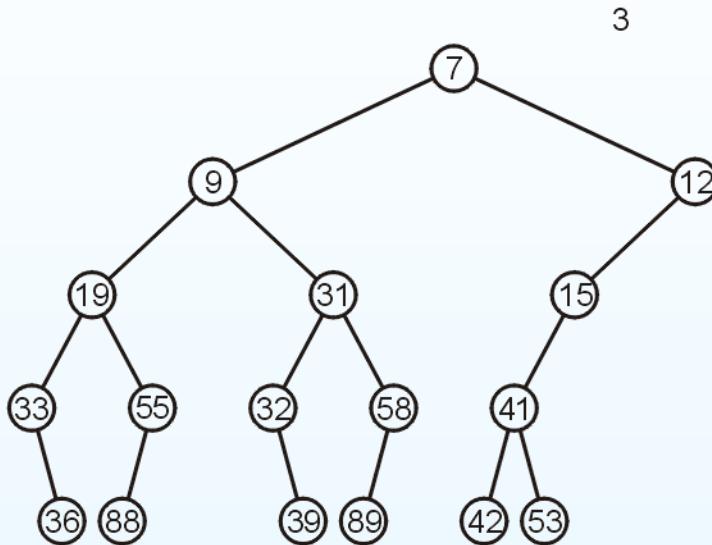
We want to remove the minimum value from the heap:



1. Remove 3
2. Promote 7 (the minimum of 7 and 12) to the root
3. Recursively, promote 9 (the minimum of 9 and 31)
4. Recursively, promote 19 (the minimum of 33 and 19)

Deletion

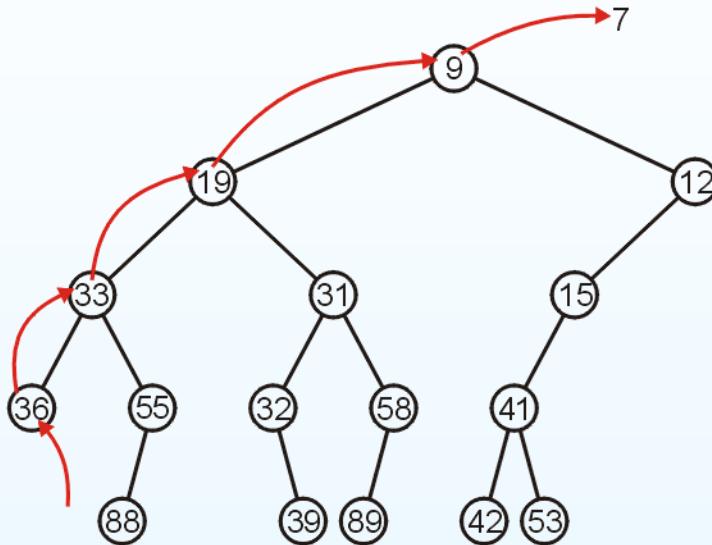
We want to remove the minimum value from the heap:



1. Remove 3
2. Promote 7 (the minimum of 7 and 12) to the root
3. Recursively, promote 9 (the minimum of 9 and 31)
4. Recursively, promote 19 (the minimum of 33 and 19)
5. Finally, 55 is a leaf node so that we promote it and delete that leaf

Deletion

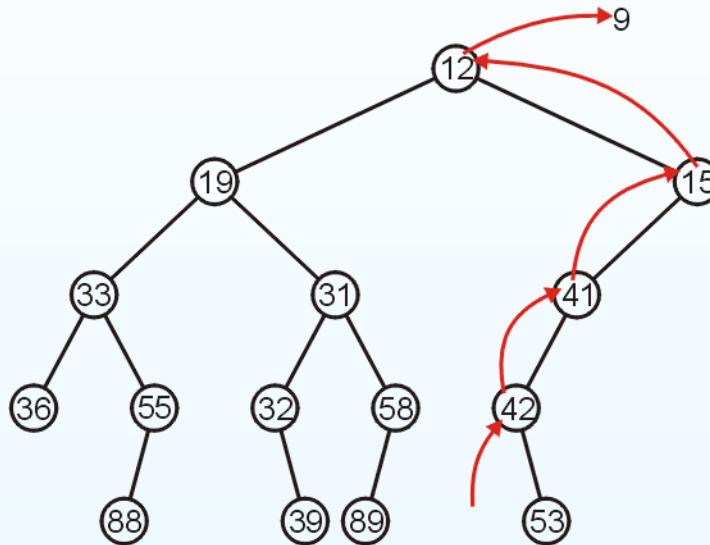
We want to remove the minimum value from the heap:



Repeat the deletion again, we can remove 7

Deletion

We want to remove the minimum value from the heap:



Again, to remove 9

Running time
 $= \Theta(h)$

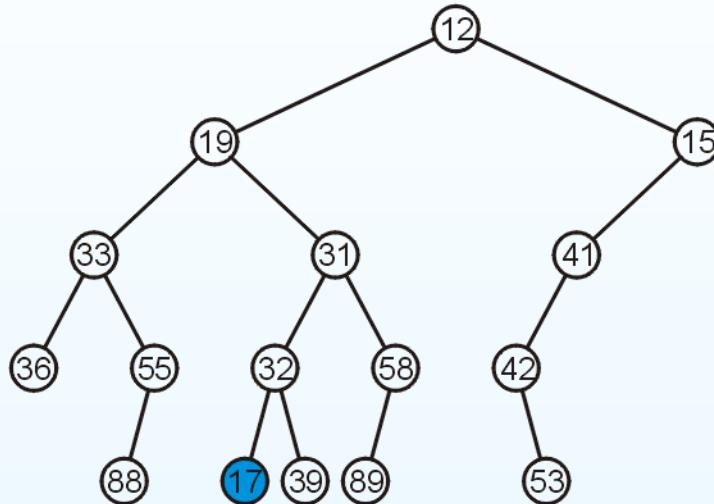
Insertion

Insert into a heap may be done either:

- At a **leaf** – move up if it is smaller than the parent
- At the **root** – move down to one of subtrees if it is larger

Insertion: Approach 1 (At a Leaf)

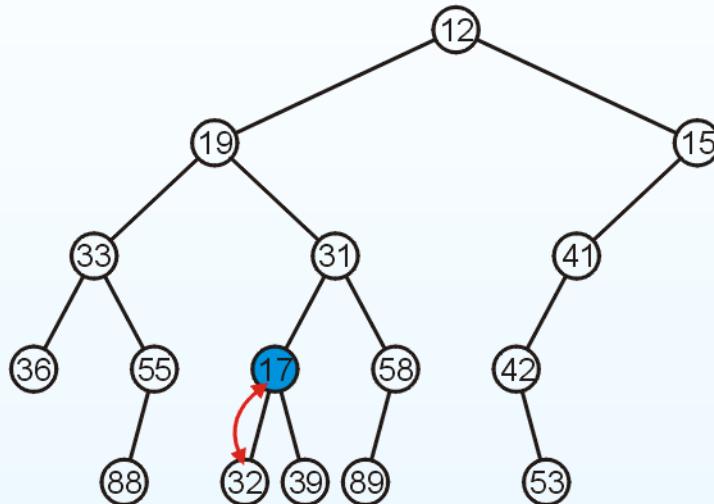
We want to insert 17 into the heap:



1. Select an arbitrary node to insert a new leaf node

Insertion: Approach 1 (At a Leaf)

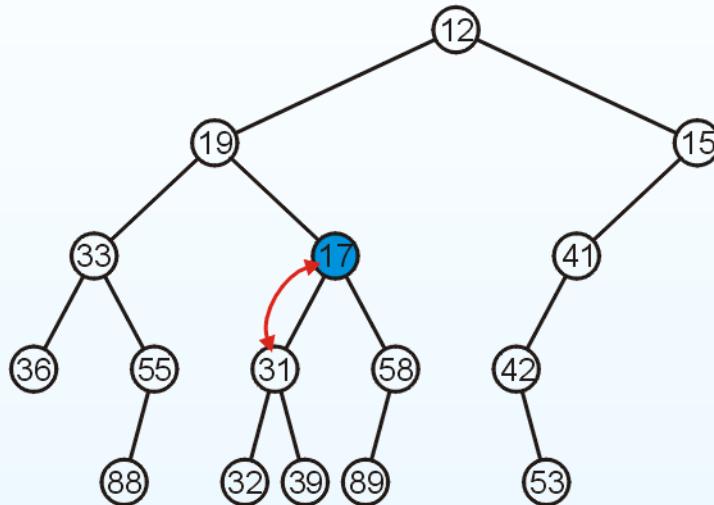
We want to insert 17 into the heap:



1. Select an arbitrary node to insert a new leaf node
2. The node 17 is less than its parent node 32, so we swap them

Insertion: Approach 1 (At a Leaf)

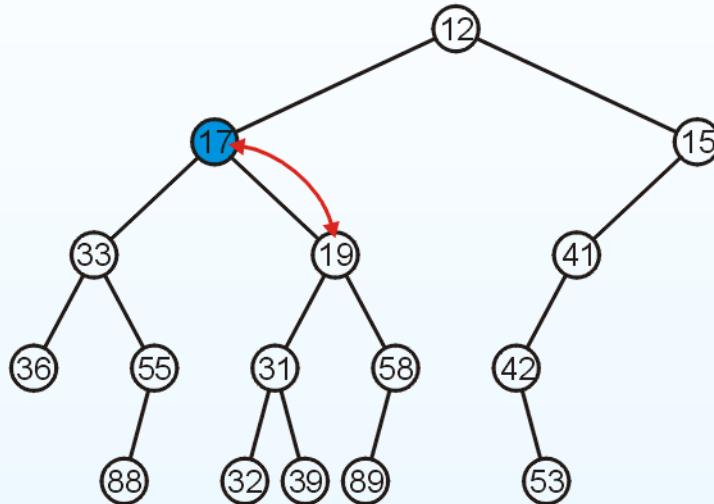
We want to insert 17 into the heap:



1. Select an arbitrary node to insert a new leaf node
2. The node 17 is less than its parent node 32, so we swap them
3. The node 17 is also less than the node 31; swap them

Insertion: Approach 1 (At a Leaf)

We want to insert 17 into the heap:

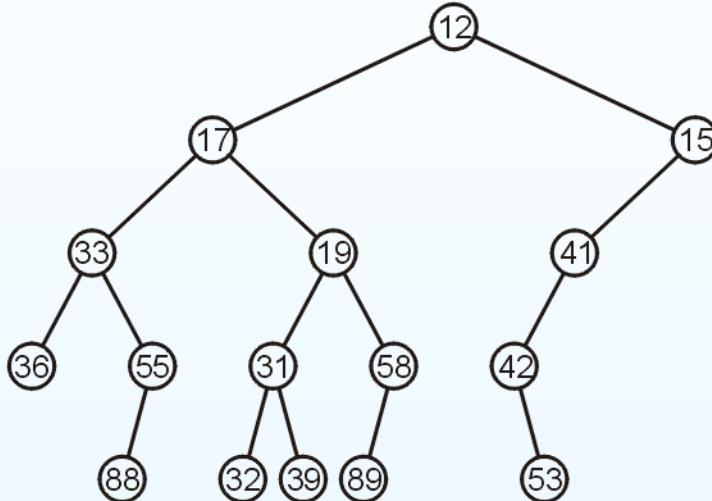


1. Select an arbitrary node to insert a new leaf node
2. The node 17 is less than its parent node 32, so we swap them
3. The node 17 is also less than the node 31; swap them
4. The node 17 is also less than the node 19; swap them

Insertion: Approach 1 (At a Leaf)

We want to insert 17 into the heap:

Running time
 $= \Theta(h)$



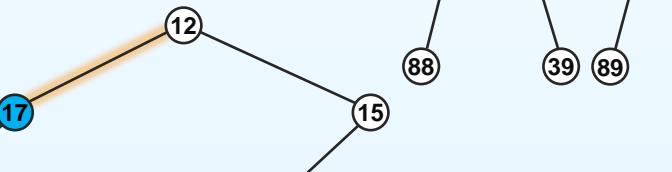
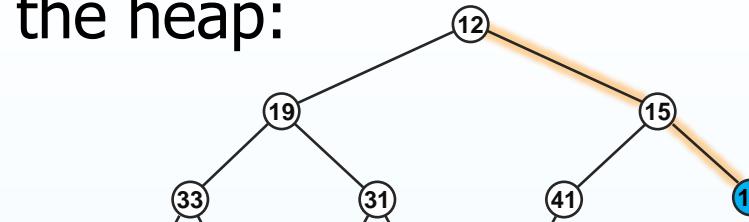
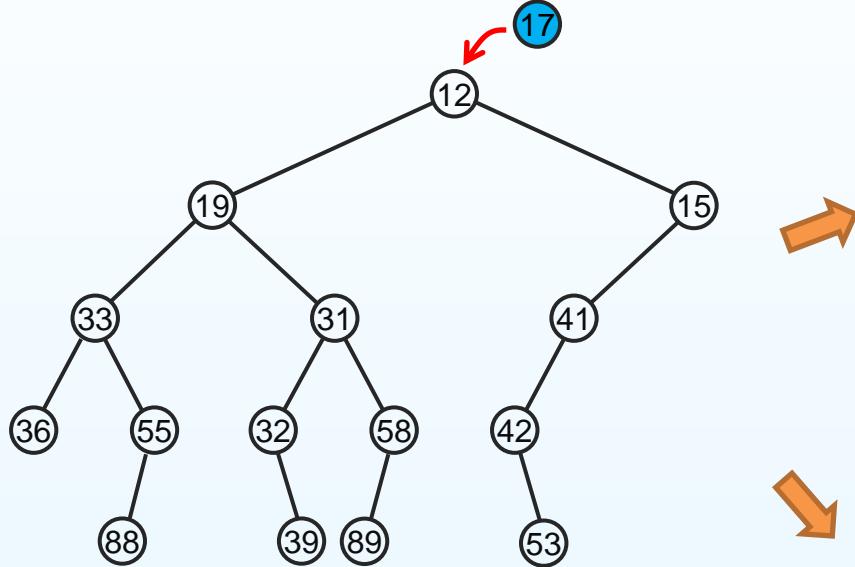
This process is called
“percolation”

1. Select an arbitrary node to insert a new leaf node
2. The node 17 is less than its parent node 32, so we swap them
3. The node 17 is also less than the node 31; swap them
4. The node 17 is also less than the node 19; swap them
5. The node 17 is greater than the node 12, so we are finished

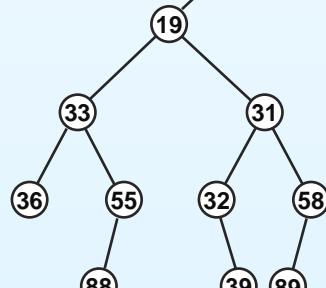
What is the problem?

Insertion: Approach 2 (At the Root)

We want to insert 17 into the heap:



What is the problem?



Implementation

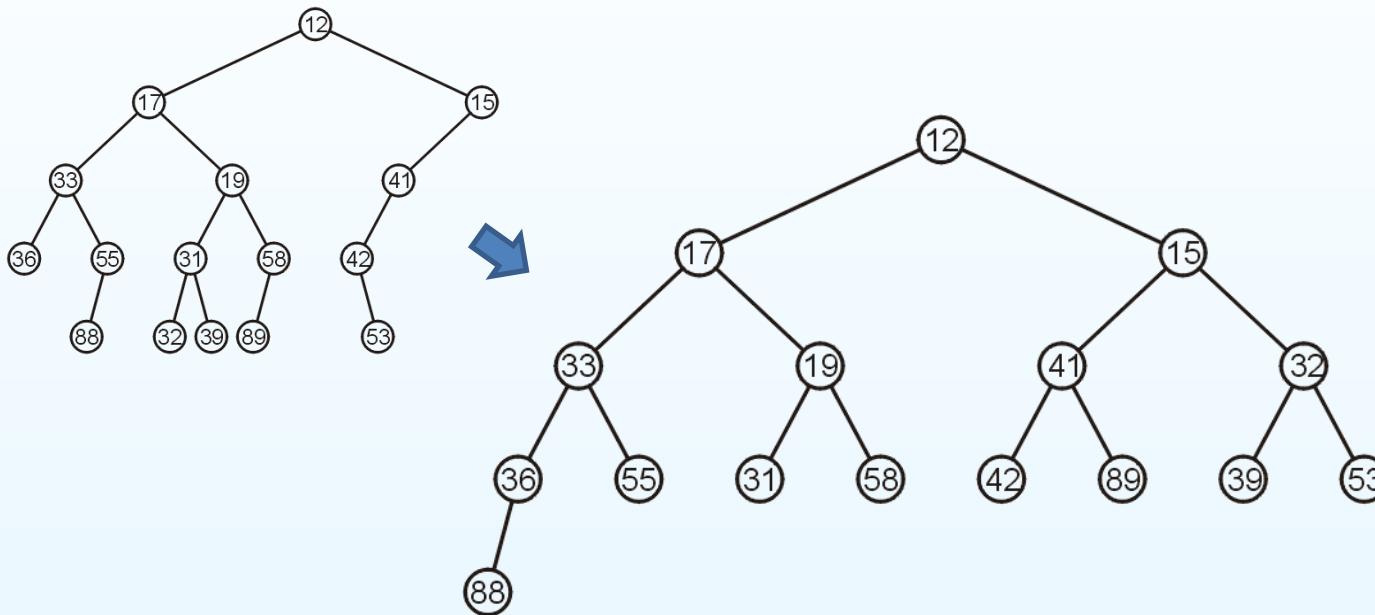
- With binary search trees, we introduced the concept of **balance** such as AVL trees
- However, to keep balance with binary heaps, there are various solutions:
 - Complete binary trees
 - Leftist heaps
 - Skew heaps
- We will look at using **complete binary trees**

Complete Binary Heaps

- Based on the concept of complete binary trees, we can maintain the **complete binary heap** structure with minimal effort
- We have already seen that a complete tree is easily stored as an **array**
 - We can store a heap of size n as an array of size $\Theta(n)$

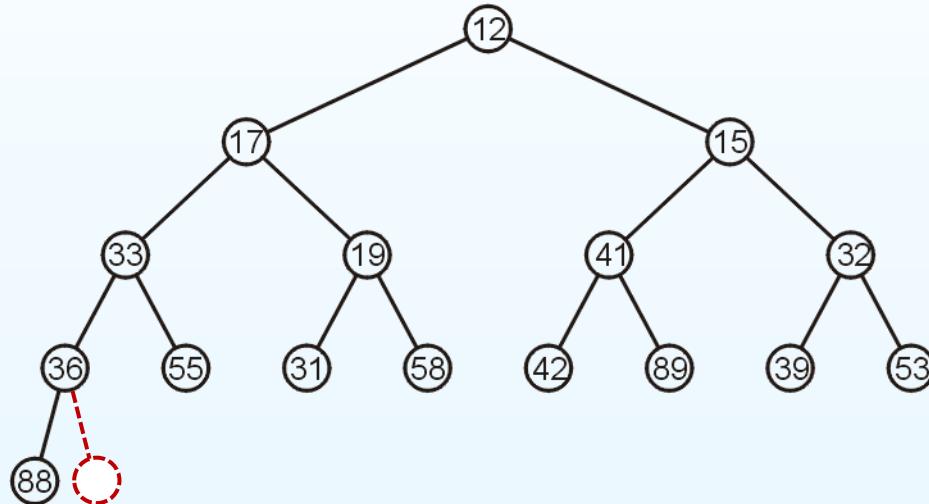
Complete Binary Heaps

For example, the previous heap may be represented as the following (**non-unique**) complete tree:



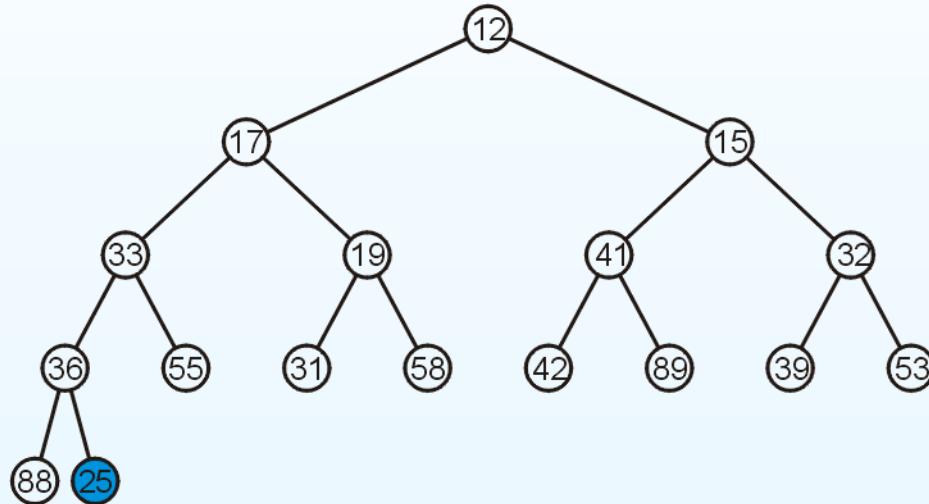
Complete Binary Heaps: Insertion

To insert, we need only place the new node as a **leaf node** in the appropriate location and percolate up



Complete Binary Heaps: Insertion

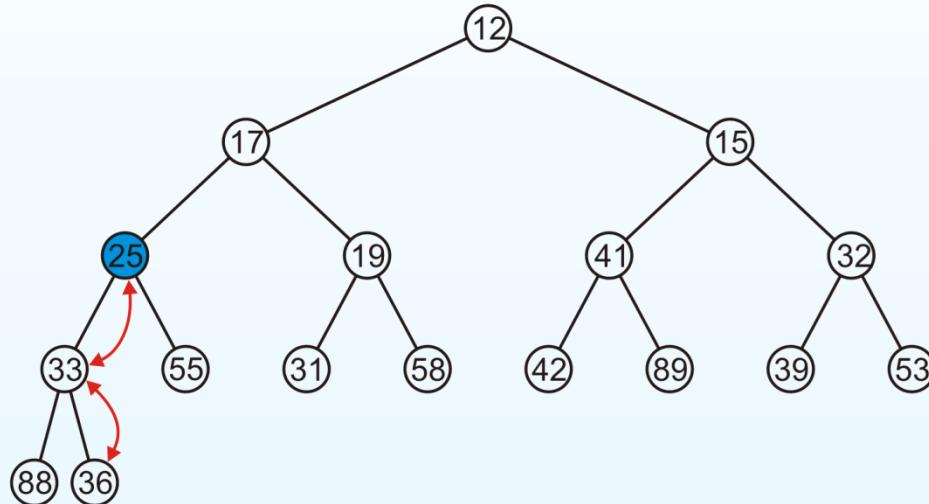
We will insert 25:



Complete Binary Heaps: Insertion

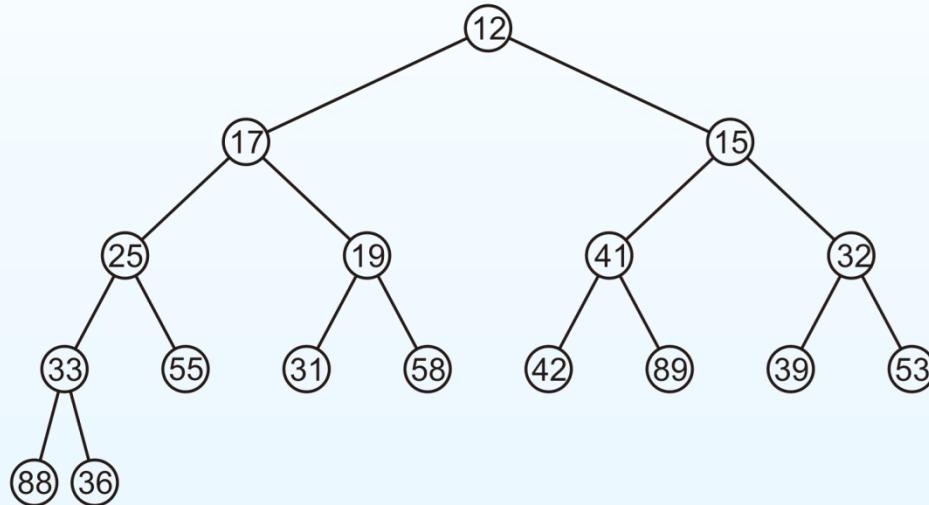
We will insert 25:

- Percolate 25 up into its appropriate location



Complete Binary Heaps: Deletion

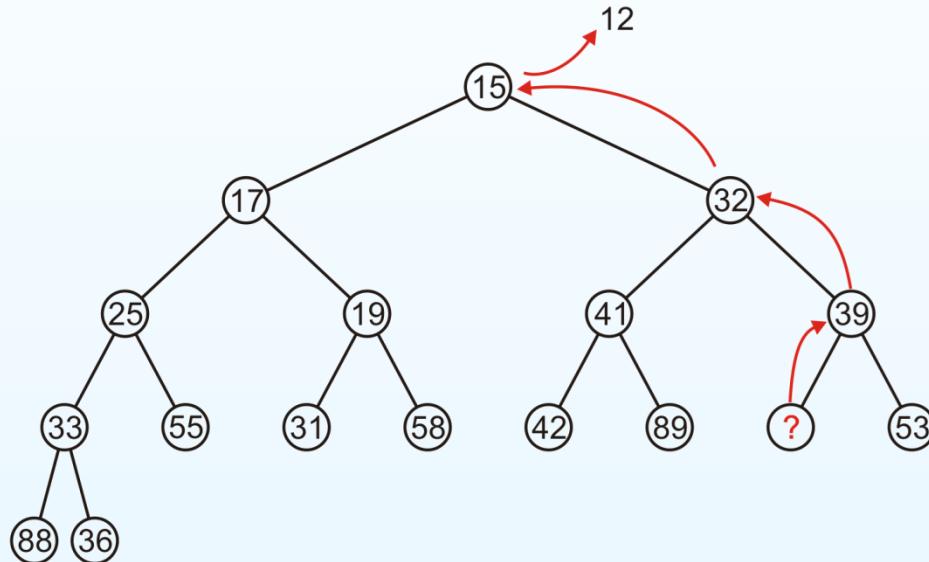
Support we want to delete the minimum value, i.e., 12:



Complete Binary Heaps: Deletion

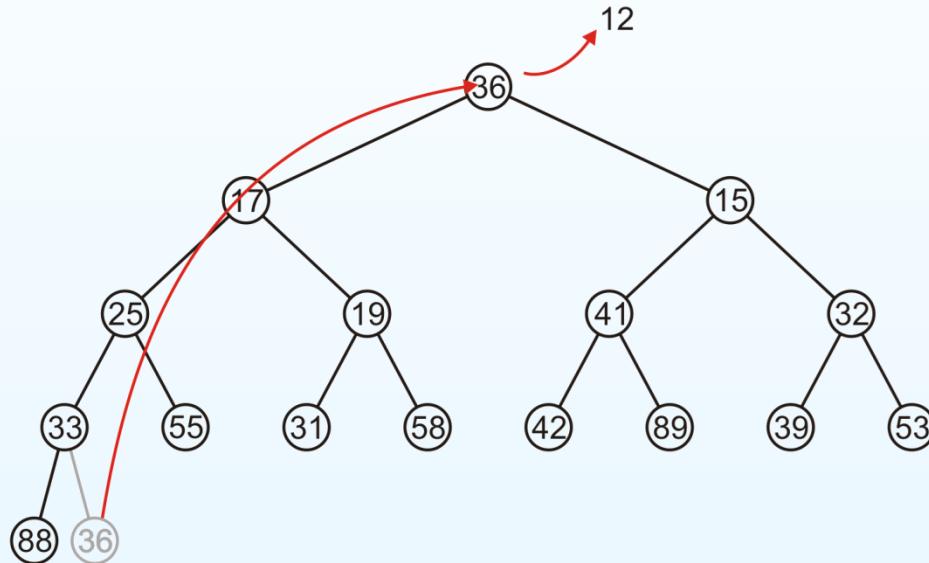
Support we want to delete the minimum value, i.e., 12:

- Percolating up will create a hole leading to a non-complete tree



Complete Binary Heaps: Deletion

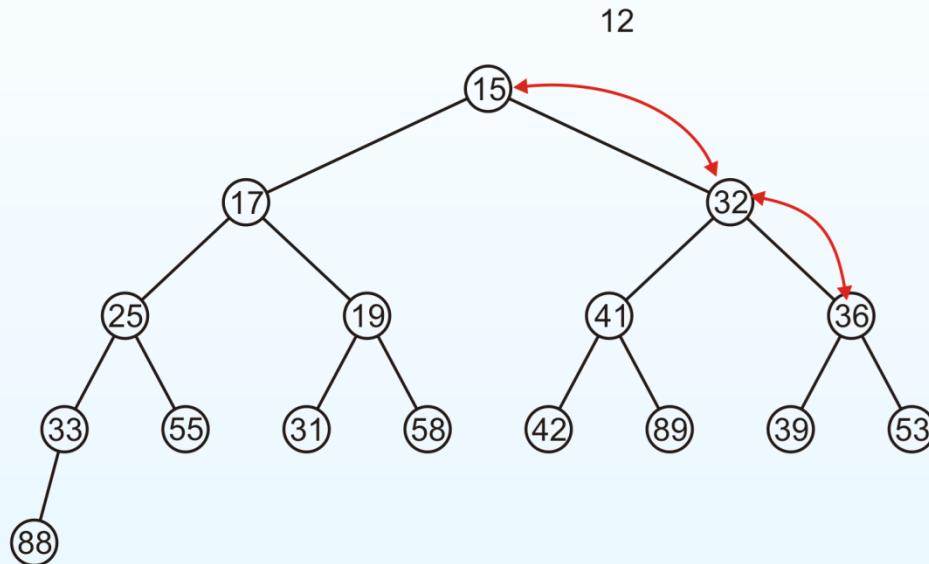
Therefore, to delete, we will copy the **last element** to the root and then **percolate down**



Complete Binary Heaps: Deletion

We delete the minimum value 12:

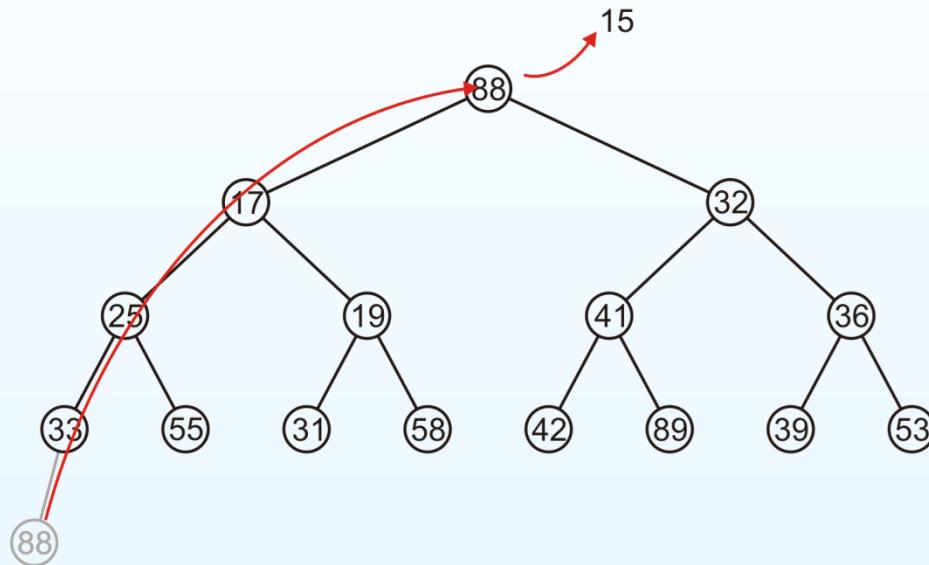
1. Copy 36 to the root
2. Percolate 36 down by swapping it with the smallest of its children



Complete Binary Heaps: Deletion

Again, delete 15:

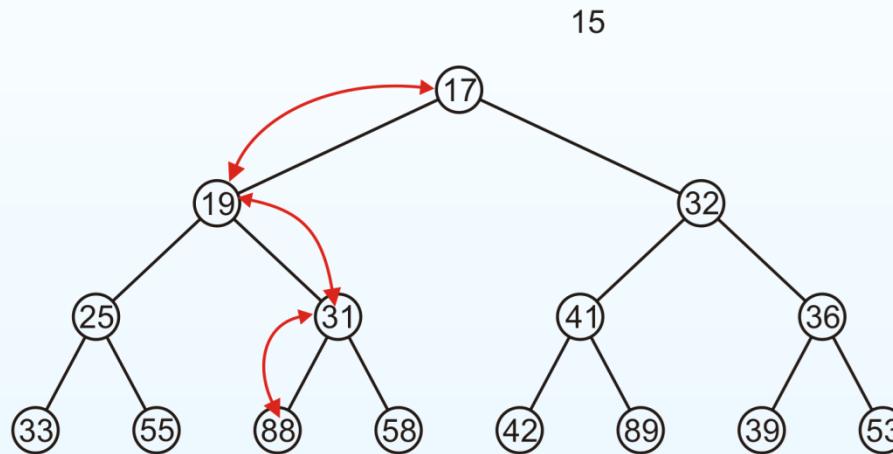
1. Copy 88 to the root



Complete Binary Heaps: Deletion

Again, delete 15:

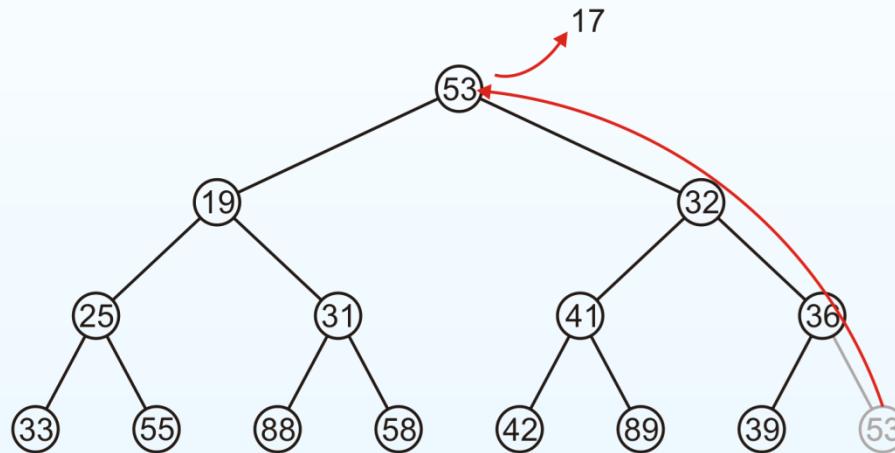
1. Copy 88 to the root
2. Percolate 88 down to the point where it has no children



Complete Binary Heaps: Deletion

Again, delete 17:

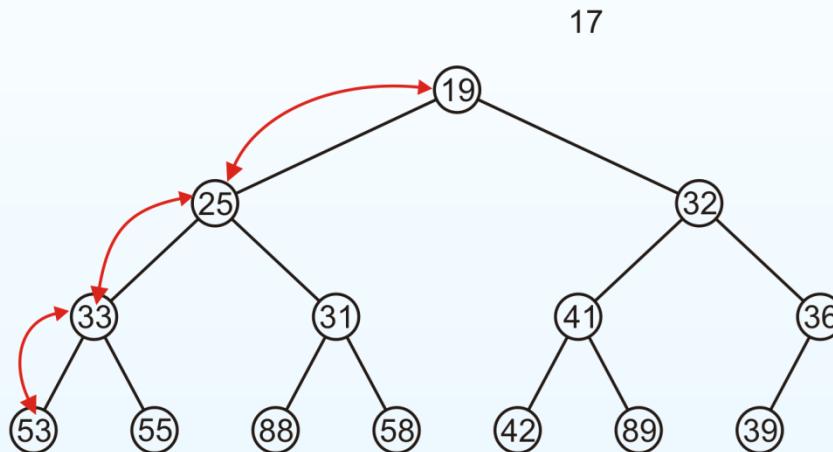
1. Copy 53 to the root



Complete Binary Heaps: Deletion

Again, delete 17:

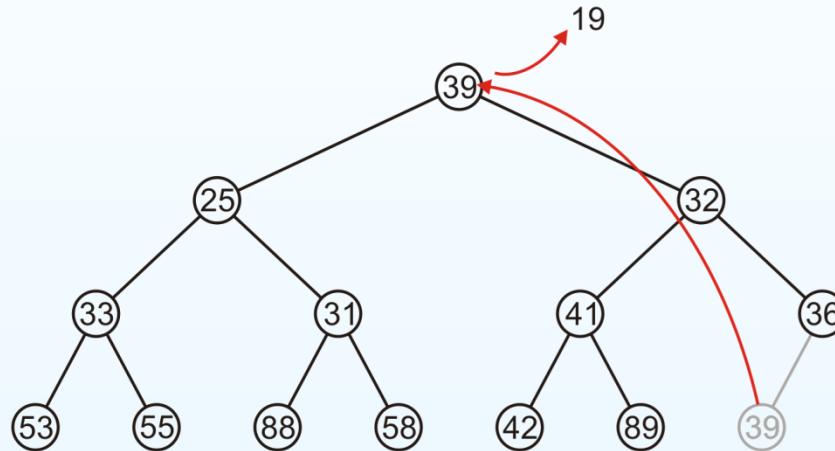
1. Copy 53 to the root
2. Percolate 53 down to the appropriate location



Complete Binary Heaps: Deletion

Again, delete 19:

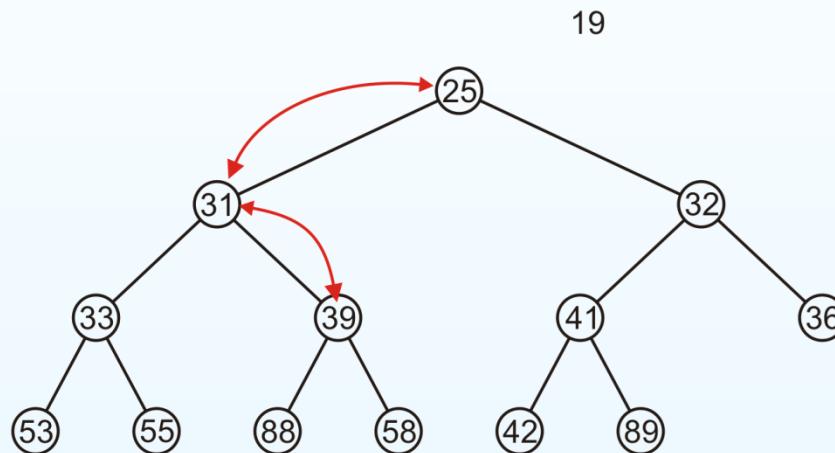
1. Copy 39 to the root



Complete Binary Heaps: Deletion

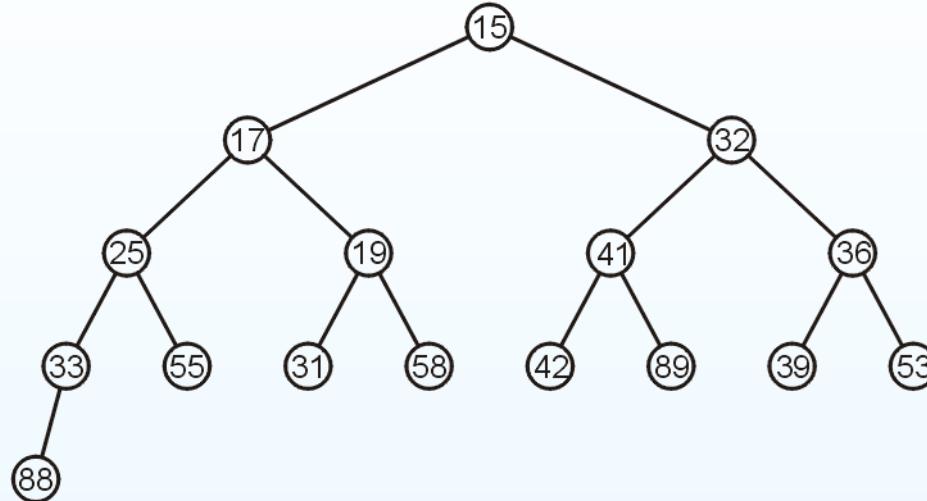
Again, delete 19:

1. Copy 39 to the root
2. Percolate 39 down to the appropriate location

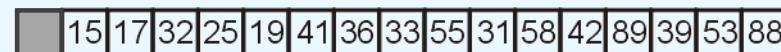


Array Implementation

For the heap:

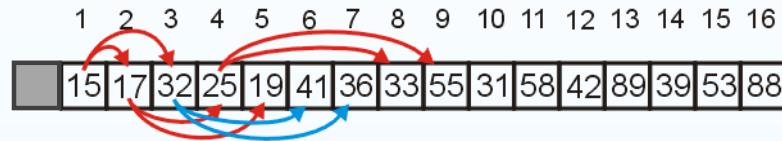


A breadth-first traversal yields:



Array Implementation

Recall that if we associate an index—starting at 1—with each element in the breadth-first traversal, we get:

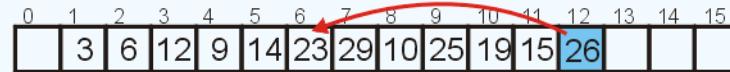
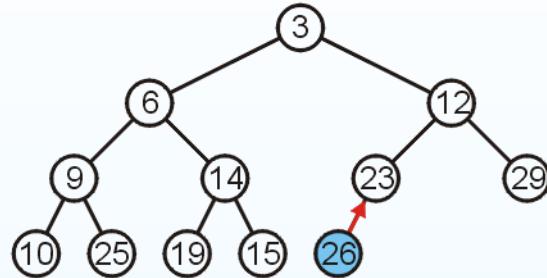


Given the element at index k , it follows that:

- The parent is at $[k/2]$
- The children are at $2k$ and $2k + 1$

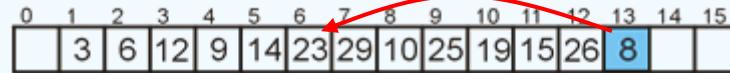
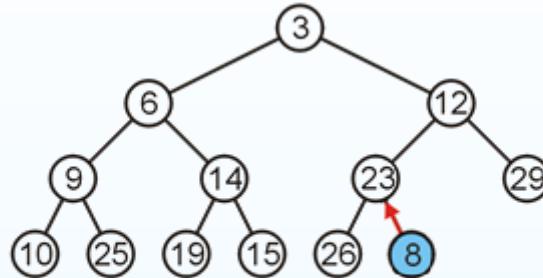
Array Implementation: Insertion

Inserting 26 requires no changes



Array Implementation: Insertion

Inserting 8 requires percolations:

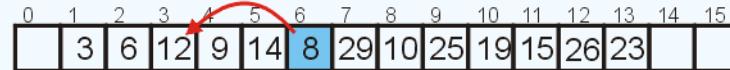
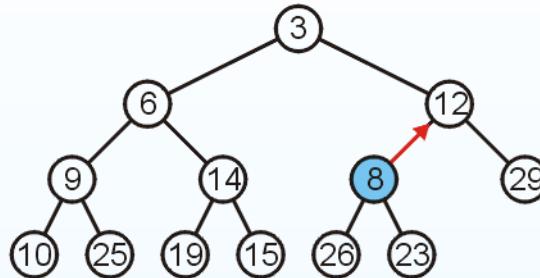


1. Swap 8 and 23



Array Implementation: Insertion

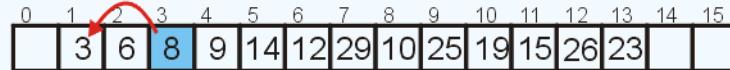
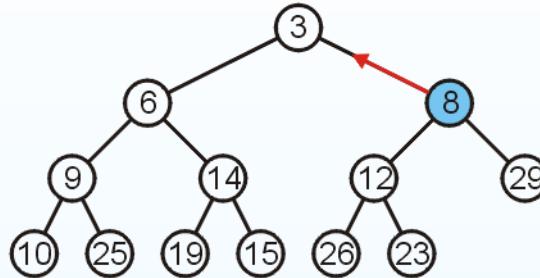
Inserting 8 requires percolations:



1. Swap 8 and 23
2. Swap 8 and 12

Array Implementation: Insertion

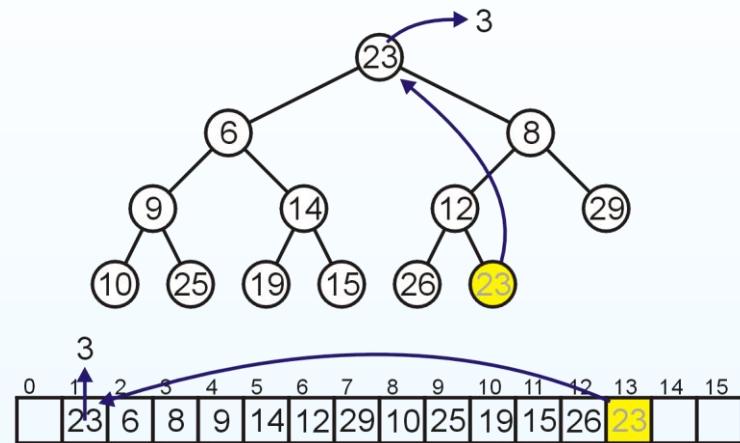
Inserting 8 requires percolations:



1. Swap 8 and 23
2. Swap 8 and 12
3. The node 8 is greater than its parent, so we are finished

Array Implementation: Deletion

Removing the minimum value:

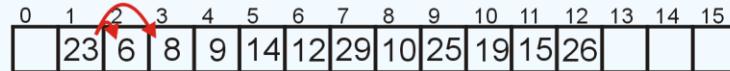
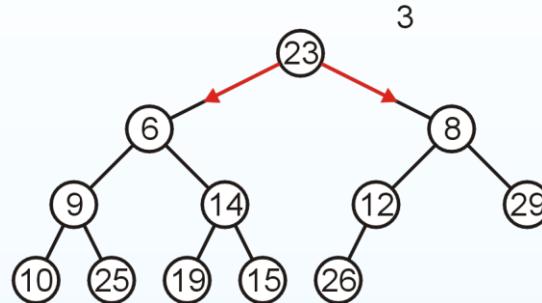


1. Copy the last element 23 to the root



Array Implementation: Deletion

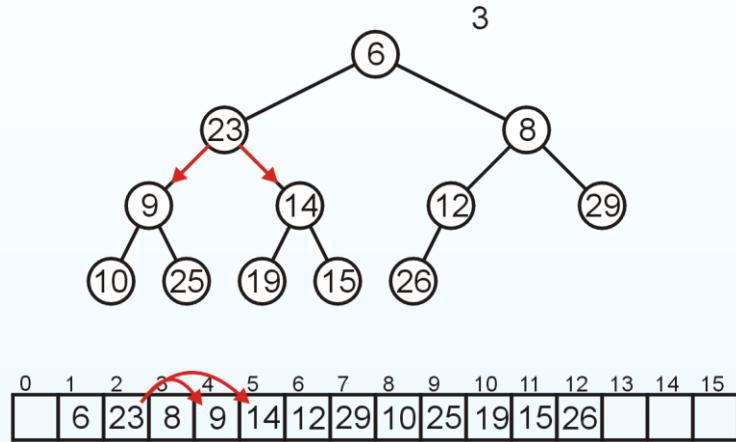
Removing the minimum value:



1. Copy the last element 23 to the root
2. Compare the node 23 (index 1) with its children (indices 2 and 3)
 - Swap 23 and 6

Array Implementation: Deletion

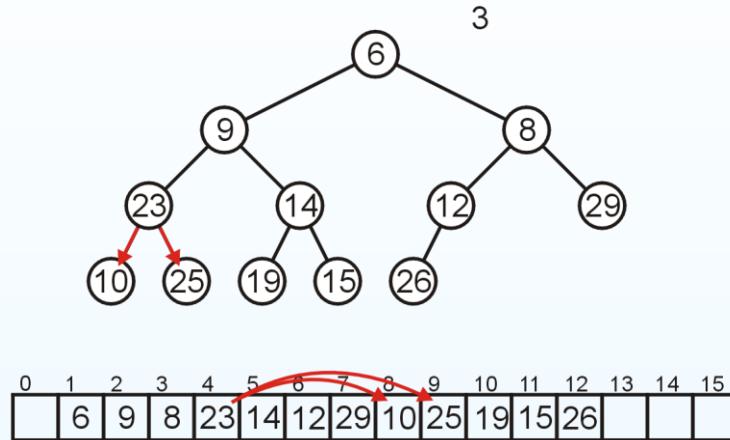
Removing the minimum value:



1. Copy the last element 23 to the root
2. Compare the node 23 (index 1) with its children (indices 2 and 3)
 - Swap 23 and 6
3. Compare the node 23 with its children: swap 23 and 9

Array Implementation: Deletion

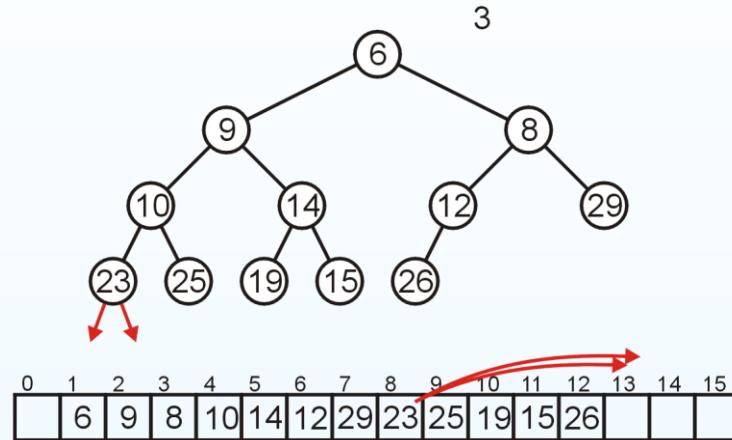
Removing the minimum value:



1. Copy the last element 23 to the root
2. Compare the node 23 (index 1) with its children (indices 2 and 3)
 - Swap 23 and 6
3. Compare the node 23 with its children: swap 23 and 9
4. Compare the node 23 with its children: swap 23 and 10

Array Implementation: Deletion

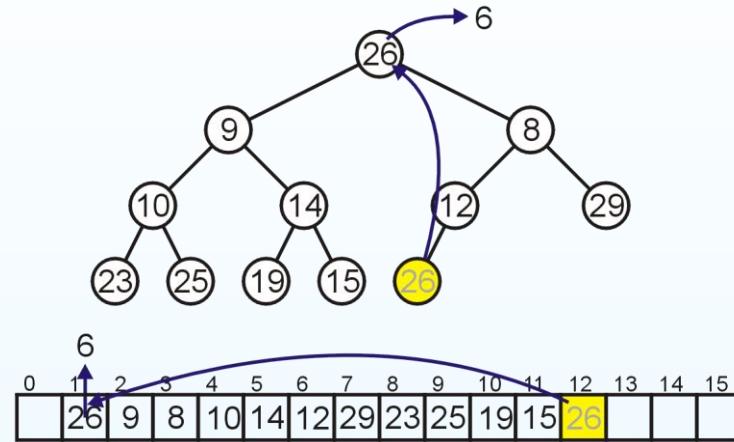
Removing the minimum value:



1. Copy the last element 23 to the root
2. Compare the node 23 (index 1) with its children (indices 2 and 3)
 - Swap 23 and 6
3. Compare the node 23 with its children: swap 23 and 9
4. Compare the node 23 with its children: swap 23 and 10
5. The node 23 is now the leaf node; stop the process

Array Implementation: Deletion

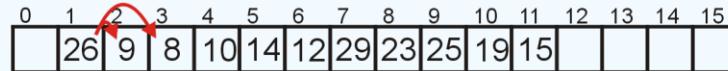
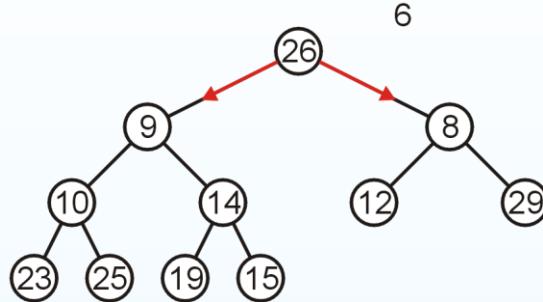
Again, removing the minimum value:



1. Copy the last element 26 to the root

Array Implementation: Deletion

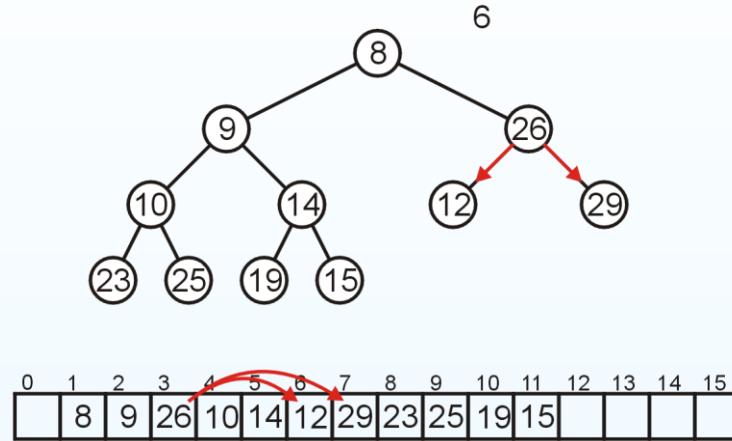
Again, removing the minimum value:



1. Copy the last element 26 to the root
2. Compare 26 with 9 and 8; swap 26 and 8

Array Implementation: Deletion

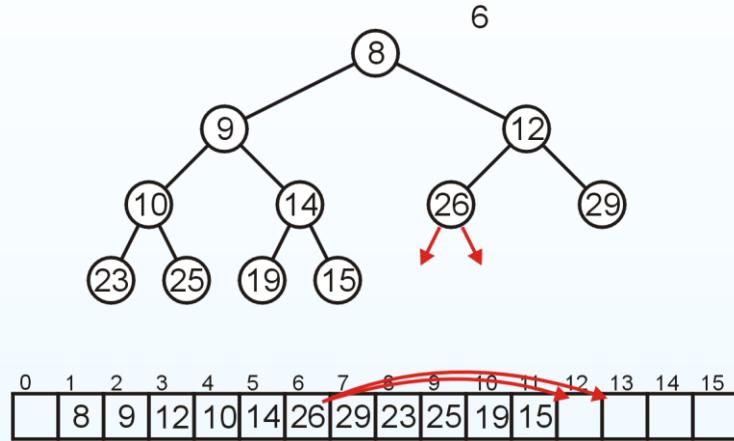
Again, removing the minimum value:



1. Copy the last element 26 to the root
2. Compare 26 with 9 and 8; swap 26 and 8
3. Compare 26 with 12 and 29; swap 26 and 12

Array Implementation: Deletion

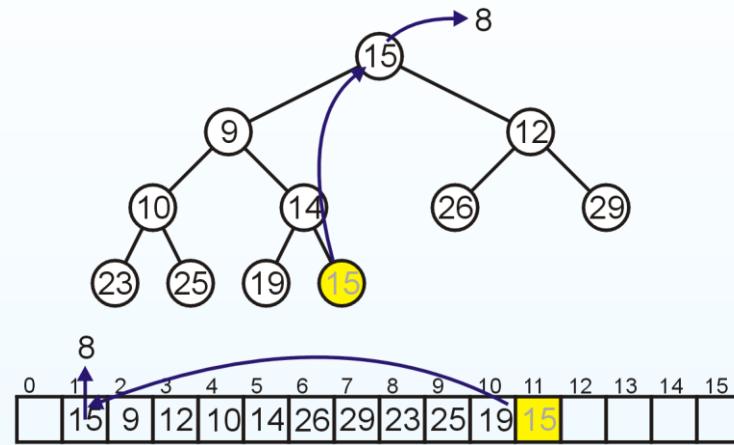
Again, removing the minimum value:



1. Copy the last element 26 to the root
2. Compare 26 with 9 and 8; swap 26 and 8
3. Compare 26 with 12 and 29; swap 26 and 12
4. The node 26 is now the leaf node; stop

Array Implementation: Deletion

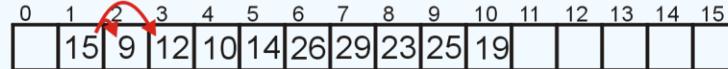
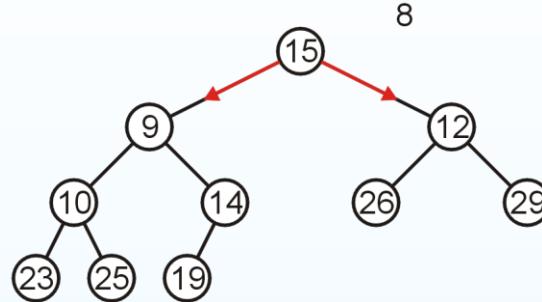
Again, removing the minimum value:



1. Copy the last element 15 to the root

Array Implementation: Deletion

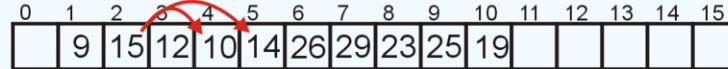
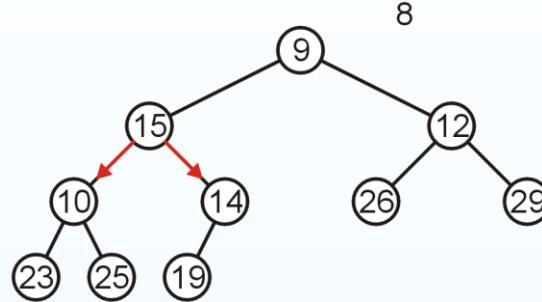
Again, removing the minimum value:



1. Copy the last element 15 to the root
2. Compare 15 with 9 and 12; swap 15 and 9

Array Implementation: Deletion

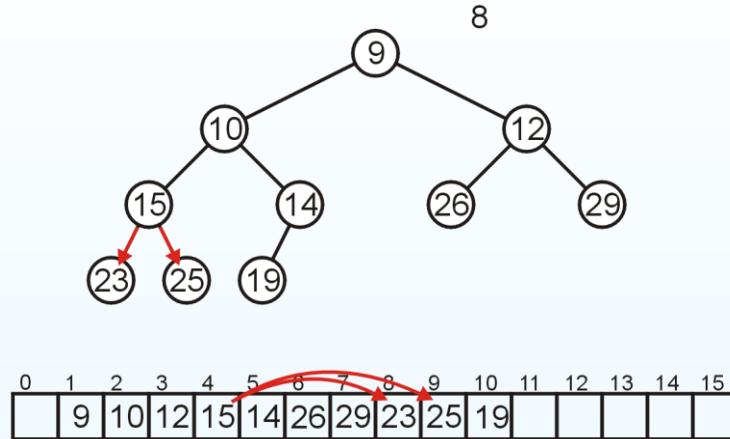
Again, removing the minimum value:



1. Copy the last element 15 to the root
2. Compare 15 with 9 and 12; swap 15 and 9
3. Compare 15 with 10 and 14; swap 15 and 10

Array Implementation: Deletion

Again, removing the minimum value:



1. Copy the last element 15 to the root
2. Compare 15 with 9 and 12; swap 15 and 9
3. Compare 15 with 10 and 14; swap 15 and 10
4. Compare 15 with 23 and 25; it is less than both so we stop

Running Time Analysis

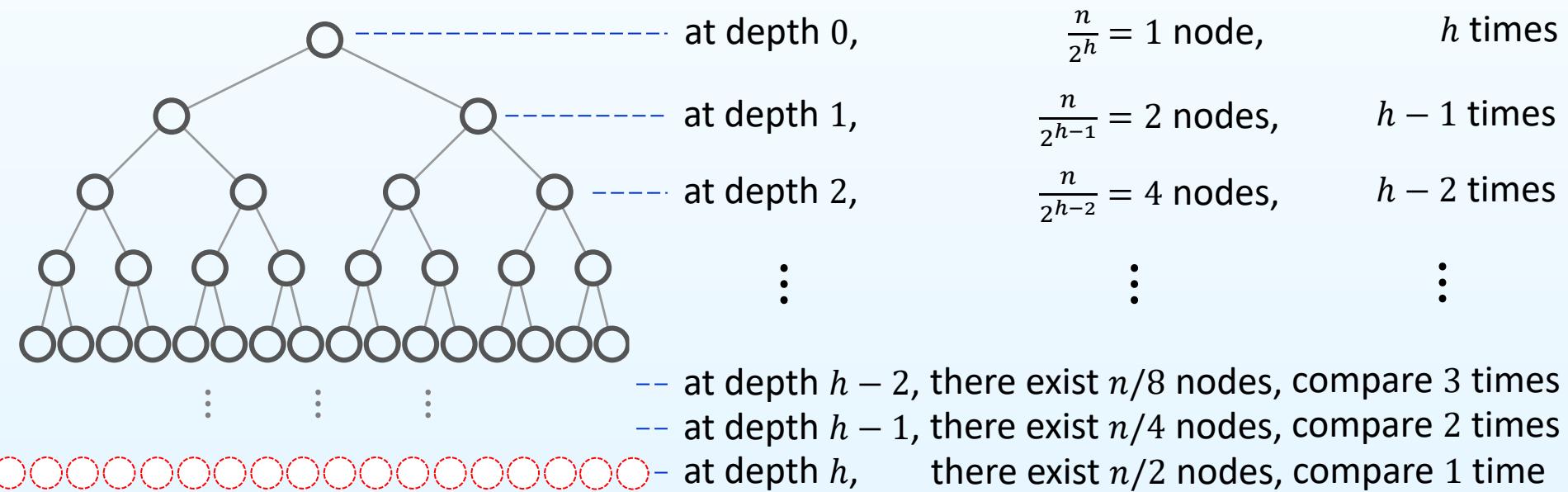
With a complete binary heap,

- The `find_min()` operation takes $\Theta(1)$
- The `delete_min()` operation takes $O(h) = O(\log n)$
 - Since we copy element (with some high value) in the lowest depth, it will likely be percolate down to the lowest depth
- The `insert()` operation takes $O(h)$... Ummm >o<



Running Time Analysis: Insertion

- If we insert object greater than any object, the best-case = $\Omega(1)$
- If we insert object less than the root, the worst-case = $O(h)$
- How about insert any value on average?
 - Inserting causes percolate up along the path to root
 - Let the heap have n nodes



$$\text{The expected (average) work to insert} = \frac{1}{2} \cdot 1 + \frac{1}{4} \cdot 2 + \frac{1}{8} \cdot 3 + \dots + \frac{1}{2^h} \cdot h = \sum_{i=1}^h \frac{i}{2^i}$$

Running Time Analysis: Insertion

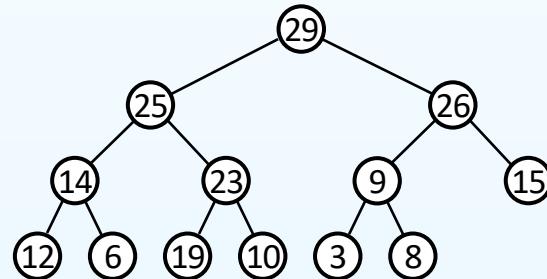
$$\begin{aligned}\sum_{i=1}^h \frac{i}{2^i} &= \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \frac{4}{16} + \cdots + \frac{h}{2^h} = x \\ \frac{1}{4} + \frac{2}{8} + \frac{3}{16} + \cdots + \frac{h-1}{2^h} + \frac{h}{2^{h+1}} &= \frac{1}{2}x \\ \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \cdots + \frac{1}{2^h} - \frac{h}{2^{h+1}} &= \frac{1}{2}x \\ \frac{\frac{1}{2} \left(1 - \left(\frac{1}{2}\right)^h\right)}{1 - \frac{1}{2}} - \frac{h}{2^{h+1}} &= \frac{1}{2}x \\ x &= \frac{2 \cdot 2^h - 2 - h}{2^h} \\ x &= \lim_{n \rightarrow \infty} \frac{2n - 2 - \log n}{n}\end{aligned}$$

the average-case for insertion = $\Theta(1)$

Binary Max Heaps

A **binary max-heap** is identical to a binary min-heap except that the parent is always larger than either of the children

For example, the same data as before stored as a complete binary max-heap yields:



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
	29	25	26	14	23	9	15	12	6	19	10	3	8		

Heaps and Priority Queues

A:	<table border="1"><tr><td></td><td>3</td><td></td><td>5</td><td></td><td>4</td><td></td><td>10</td><td></td><td>8</td><td></td><td>6</td><td></td><td>9</td><td></td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>		3		5		4		10		8		6		9		1	2	3	4	5	6	7	8						
	3		5		4		10		8		6		9																	
1	2	3	4	5	6	7	8																							

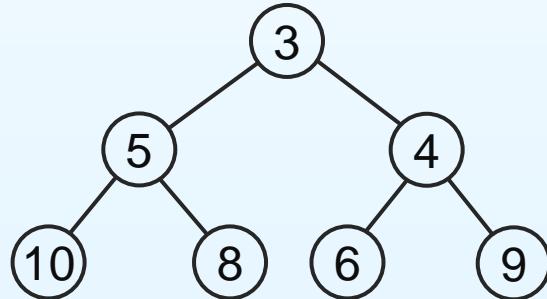
The smaller the number,
The higher the priority

B:	<table border="1"><tr><td></td><td>10</td><td></td><td>8</td><td></td><td>9</td><td></td><td>6</td><td></td><td>5</td><td></td><td>3</td><td></td><td>4</td><td></td></tr><tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td></td><td></td><td></td><td></td><td></td><td></td></tr></table>		10		8		9		6		5		3		4		1	2	3	4	5	6	7	8						
	10		8		9		6		5		3		4																	
1	2	3	4	5	6	7	8																							

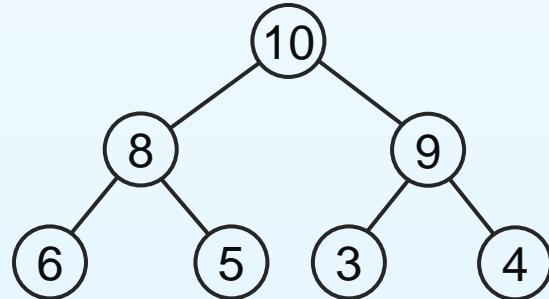
The larger the number,
The higher the priority

- Operations on a traditional array may take $O(n)$
- While operations on a heap take $O(\log n)$

Min Heap



Max Heap



Outline

- Priority Queues
- Binary Heaps
- Application:
 - Huffman code

Problem

Suppose we want to encode a message “DEAACAAAAABA”, constructed from 5 different symbols

How many bits are required?

- Use a **fixed-length ASCII code**:
 - Each symbol requires 8 bits
 - There are 12 symbols
 - So, the message requires **96 bits**
- Use a **fixed-length Unicode**:
 - The message requires **192 bits**

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	'
1	1	1	!	33	21	41	!"	65	41	101	A	97	61	141	a
2	2	2	"	34	22	42	"	66	42	102	B	98	62	142	b
3	3	3	#	35	23	43	#	67	43	103	C	99	63	143	c
4	4	4	\$	36	24	44	\$	68	44	104	D	100	64	144	d
5	5	5	%	37	25	45	%	69	45	105	E	101	65	145	e
6	6	6	&	38	26	46	&	70	46	106	F	102	66	146	f
7	7	7	'	39	27	47	'	71	47	107	G	103	67	147	g
8	8	10	(40	28	50	(72	48	110	H	104	68	150	h
9	9	11)	41	29	51)	73	49	111	I	105	69	151	i
10	A	12	*	42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13	+	43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14	,	44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15	-	45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16	.	46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17	/	47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20	0	48	30	60	0	80	50	120	P	112	70	160	p
17	11	21	1	49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22	2	50	32	62	2	82	52	122	R	114	72	162	r
19	13	23	3	51	33	63	3	83	53	123	S	115	73	163	s
20	14	24	4	52	34	64	4	84	54	124	T	116	74	164	t
21	15	25	5	53	35	65	5	85	55	125	U	117	75	165	u
22	16	26	6	54	36	66	6	86	56	126	V	118	76	166	v
23	17	27	7	55	37	67	7	87	57	127	W	119	77	167	w
24	18	30	8	56	38	70	8	88	58	130	X	120	78	170	x
25	19	31	9	57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32	:	58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33	;	59	3B	73	;	91	5B	133	{	123	7B	173	{
28	1C	34	<	60	3C	74	<	92	5C	134	\	124	7C	174	\
29	1D	35	=	61	3D	75	=	93	5D	135	}	125	7D	175	}
30	1E	36	>	62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37	?	63	3F	77	?	95	5F	137	-	127	7F	177	-

Problem

Suppose we want to encode a message “DEAACAAAAABA”, constructed from 5 different symbols

How many bits are required?

- Use a fixed-length user-defined code:
 - 2 bits are not enough for each symbol
 - At least 3 bits are required
 - So, the message with 12 symbols will requires 36 bits

Symbol	Code
A	000
B	001
C	010
D	011
E	100

Drawbacks of Fixed-Length Codes

- Wasted space
 - There are a lot of codes that are not used for symbols
 - Unicode uses twice as much space as ASCII
 - Inefficient for plain-text messages containing only ASCII characters
- **Potential solution:** use **variable-length codes**

Variable-Length Codes

- When frequency of occurrence is known,
 - Variable number of bits to represent characters
 - Short codes for characters that occur frequently
 - On average, the length of the encoded message is less than fixed-length encoding
- **Potential problem:** how do we know where one-character ends, and another begins?
 - Not a problem if number of bits is fixed

encoded message: 001011011100010



decoded message: BDDEC

Symbol	Code
A	000
B	001
C	010
D	011
E	100

Prefix Property

- The following code does not have prefix property

Symbol	Code
A	0
B	1
C	01
D	10
E	11

encoded message: 1110



decoded message: BBBA or BEA or BBD or ED

- The **prefix property**: character code **is not** prefix of another character

Symbol	Code
A	000
B	11
C	01
D	001
E	10

encoded message: 01001101100010



decoded message: CDEBAE

Problem

Suppose we want to encode a message “DEAACAAAAABA”, constructed from 5 different symbols

How many bits are required?

- Use a variable-length code:

Symbol	Frequency	Code
A	8	0
B	1	01 1 10
C	1	110
D	1	1110
E	1	11110

decoded message: DEAACAAAAABA



encoded message: 

Require 22 bits

Problem

Suppose we want to encode a message “DEAACAAAAABA”, constructed from 5 different symbols

How many bits are required?

- Use a variable-length code:

Better code

Symbol	Frequency	Code
A	8	0
B	1	10
C	1	110
D	1	1110
E	1	1111

decoded message: DEAACAAAAABA



encoded message:  111011110011000000100

Require 21 bits

Problem

Suppose we want to encode a message “DEAACAAAAABA”, constructed from 5 different symbols

How many bits are required?

- Use a variable-length code:

The best code

Symbol	Frequency	Code
A	8	0
B	1	100
C	1	101
D	1	110
E	1	111

decoded message: DEAACAAAAABA



encoded message: 

Require 20 bits

Huffman Coding Trees

- Binary tree
 - Each leaf contains symbol (character)
 - Label edge from node to **left child with 0**
 - Label edge from node to **right child with 1**
- Code for any symbol obtained by following **path** from the root to the leaf node containing symbol
- Code has the **prefix property**
 - Leaf node cannot appear on path to another leaf

Building a Huffman tree

- Find frequencies of each symbol occurring in message
- Begin with a forest of single node trees
 - Each contains symbol and its frequency
- Do repeatedly
 - Select two trees with the smallest frequency at the root
 - Produce a new binary tree with the selected trees as children and store the sum of their frequencies in the root
- Repetition end when there is one tree
 - This is the Huffman coding tree

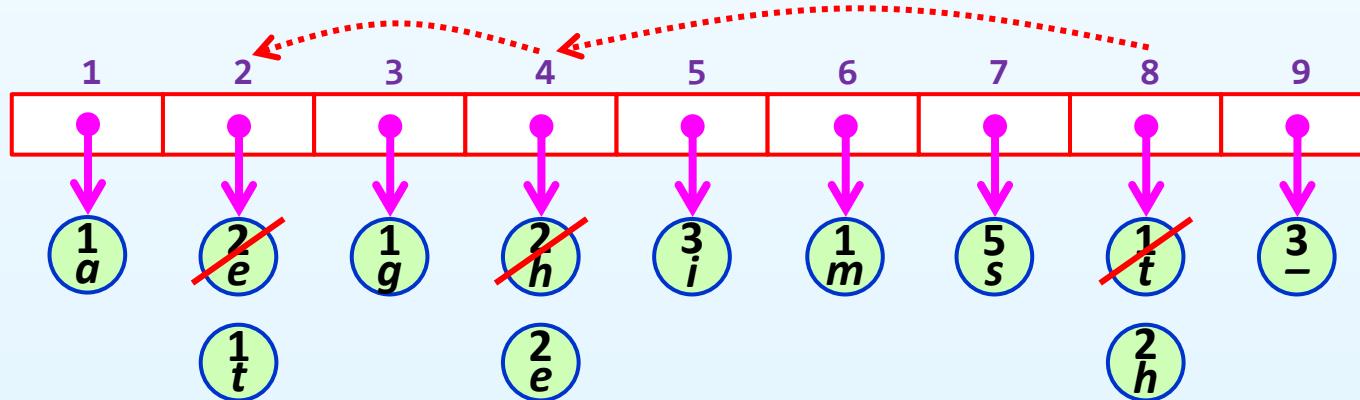
Example

- Build the Huffman coding tree for the message:
this is his message

- Character frequencies

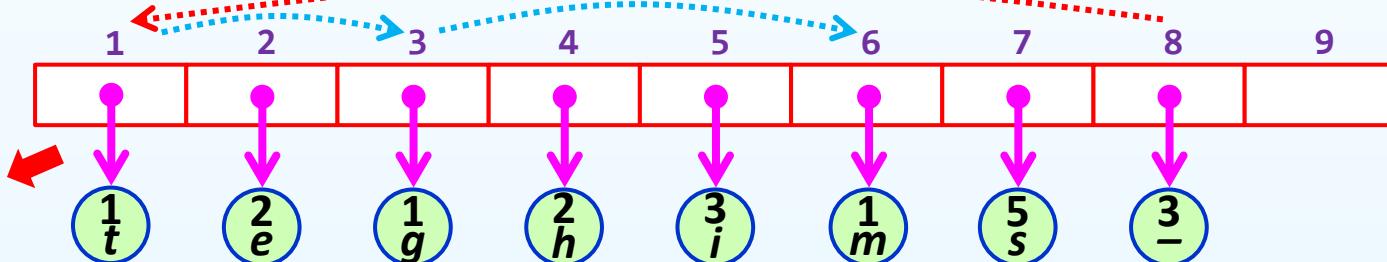
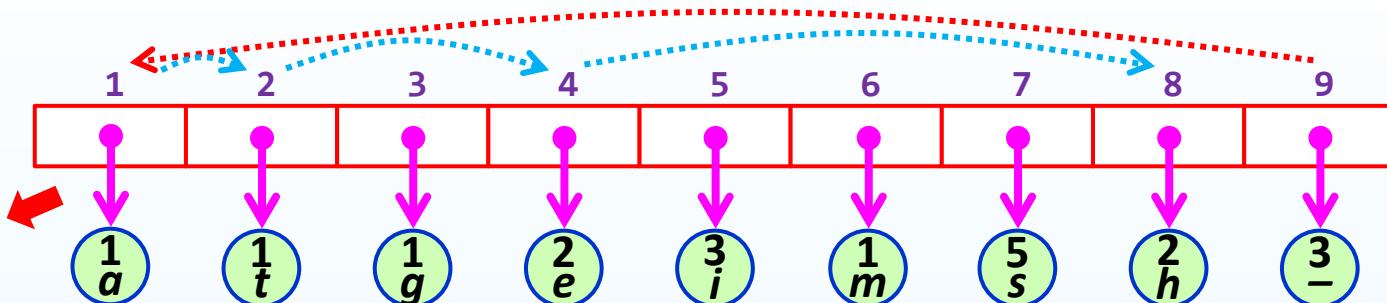
a	e	g	h	i	m	s	t	_
1	2	1	2	3	1	5	1	3

- Begin with a forest of single trees representing by a priority queue (min-heap)

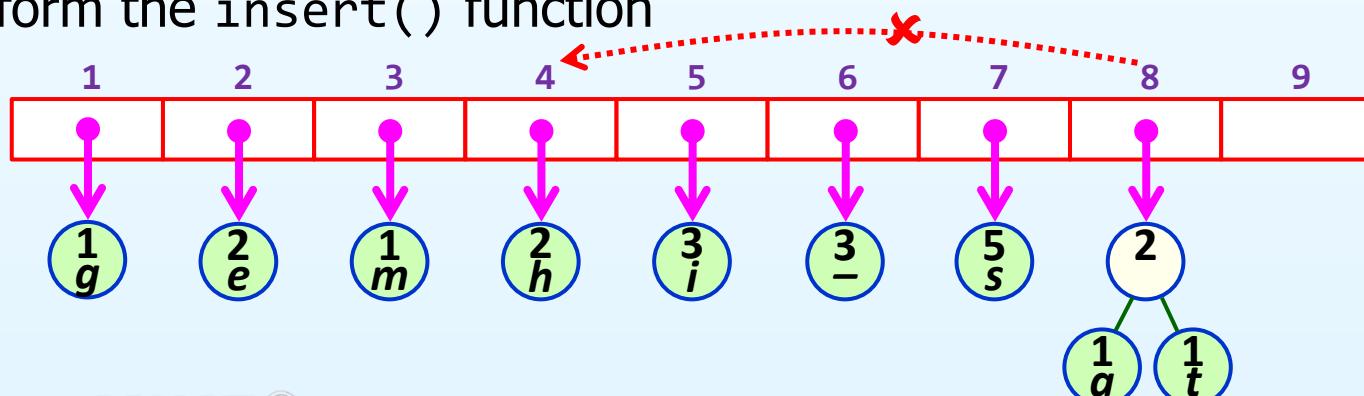


Example: Round 1

- Perform the `delete_min()` function 2 times

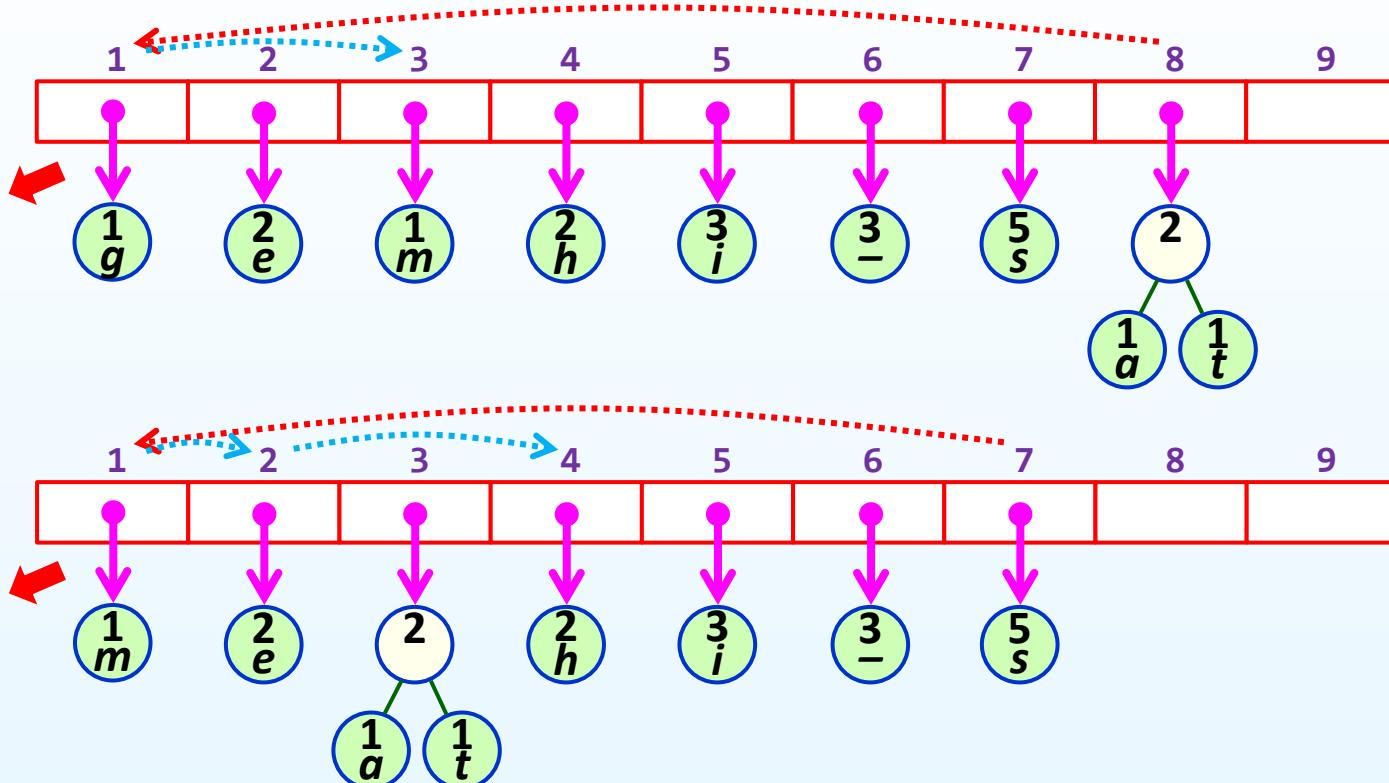


- Perform the `insert()` function



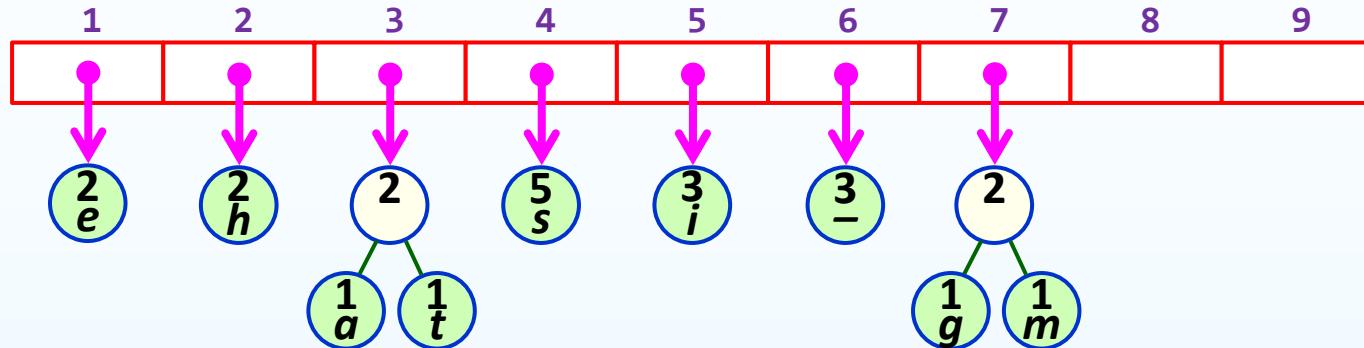
Example: Round 2

- Perform the `delete_min()` function 2 times



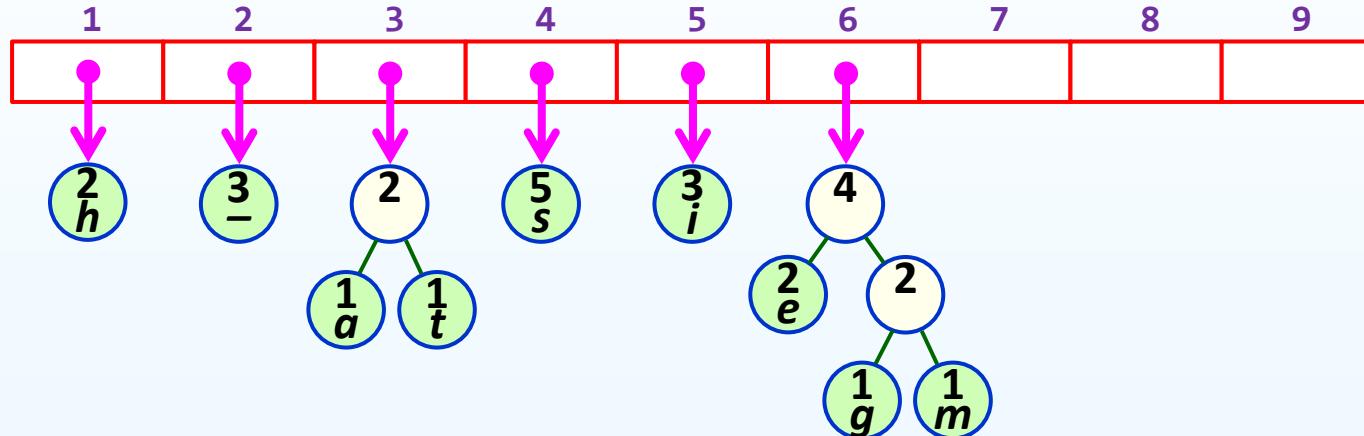
Example: Round 2

- Perform the insert() function



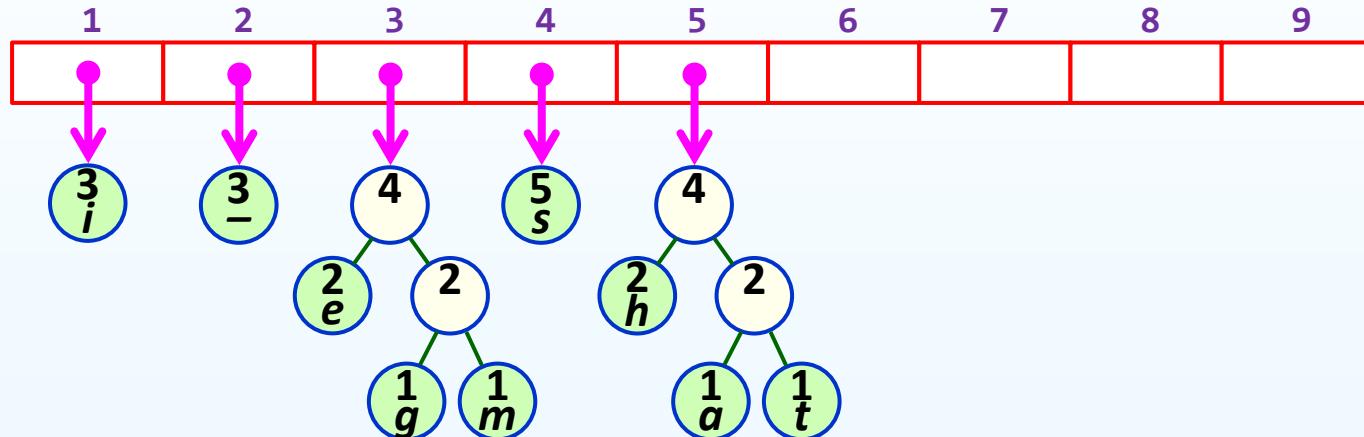
Example: Round 3

- Perform the `delete_min()` function 2 times
- Perform the `insert()` function



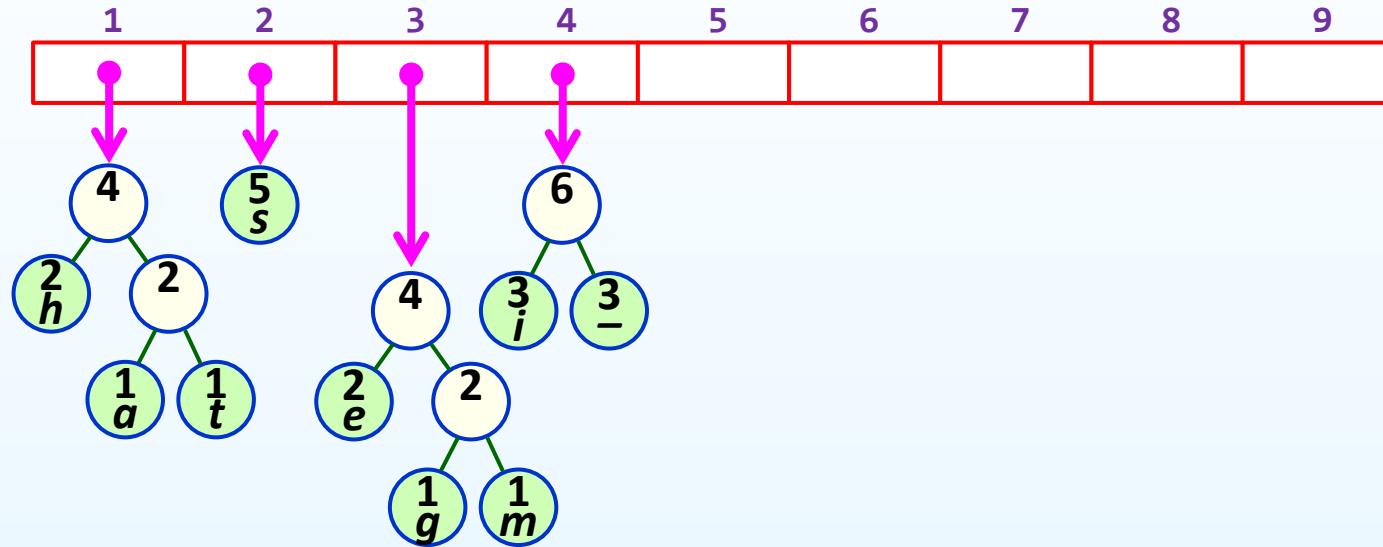
Example: Round 4

- Perform the `delete_min()` function 2 times
- Perform the `insert()` function



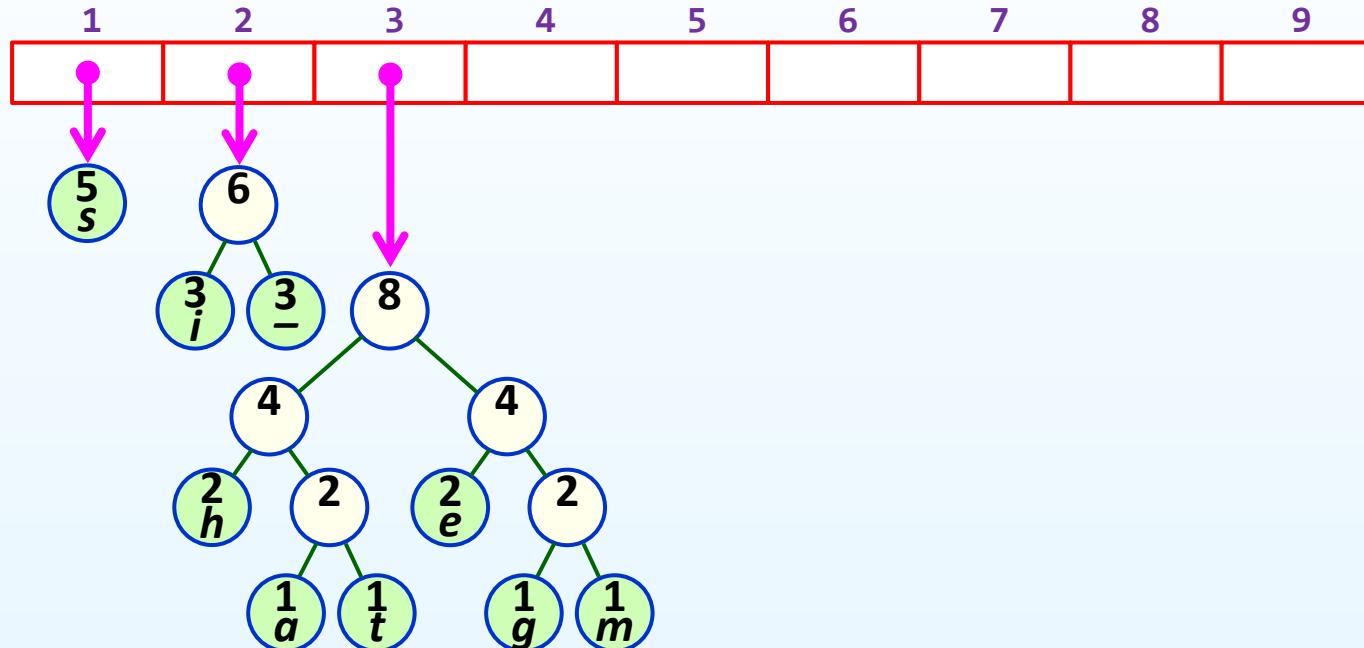
Example: Round 5

- Perform the `delete_min()` function 2 times
- Perform the `insert()` function



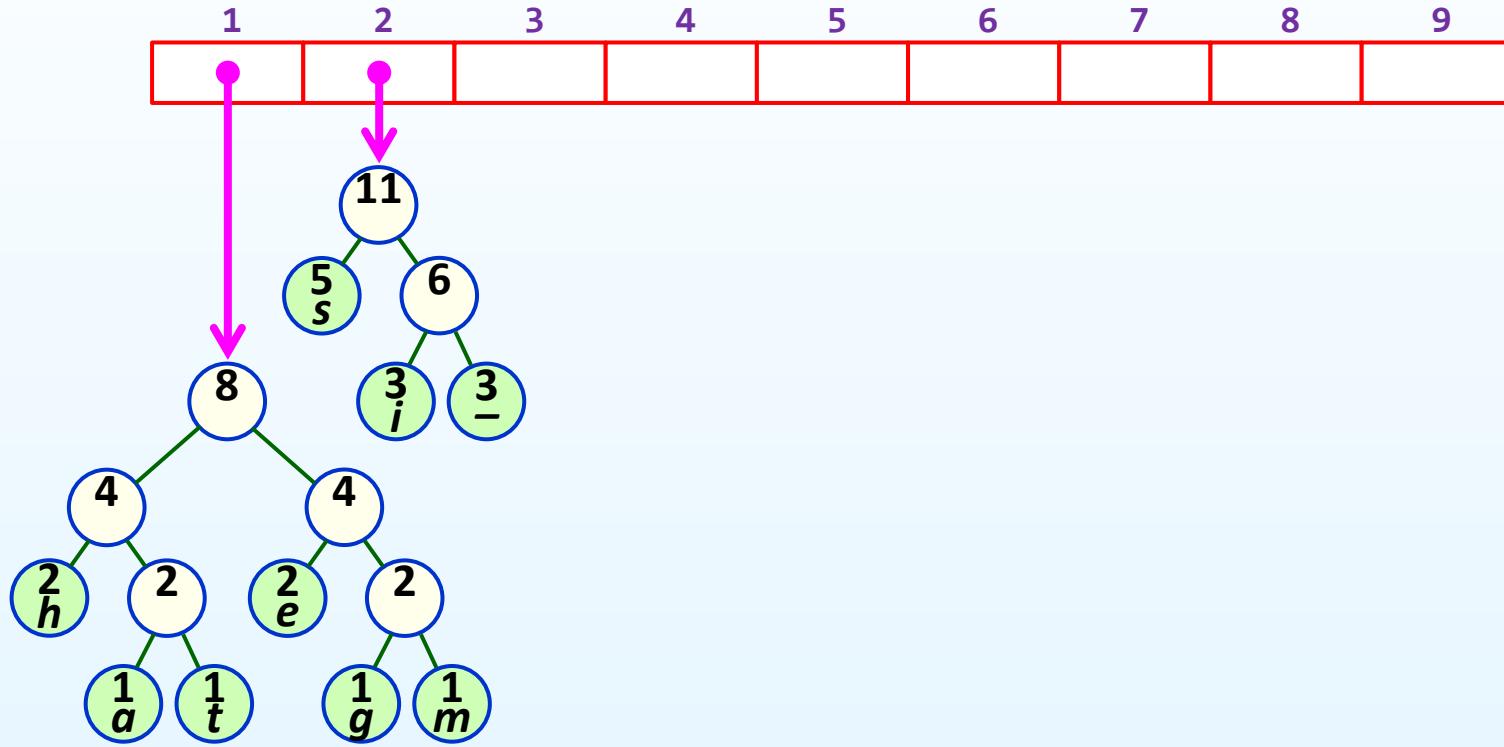
Example: Round 6

- Perform the `delete_min()` function 2 times
- Perform the `insert()` function



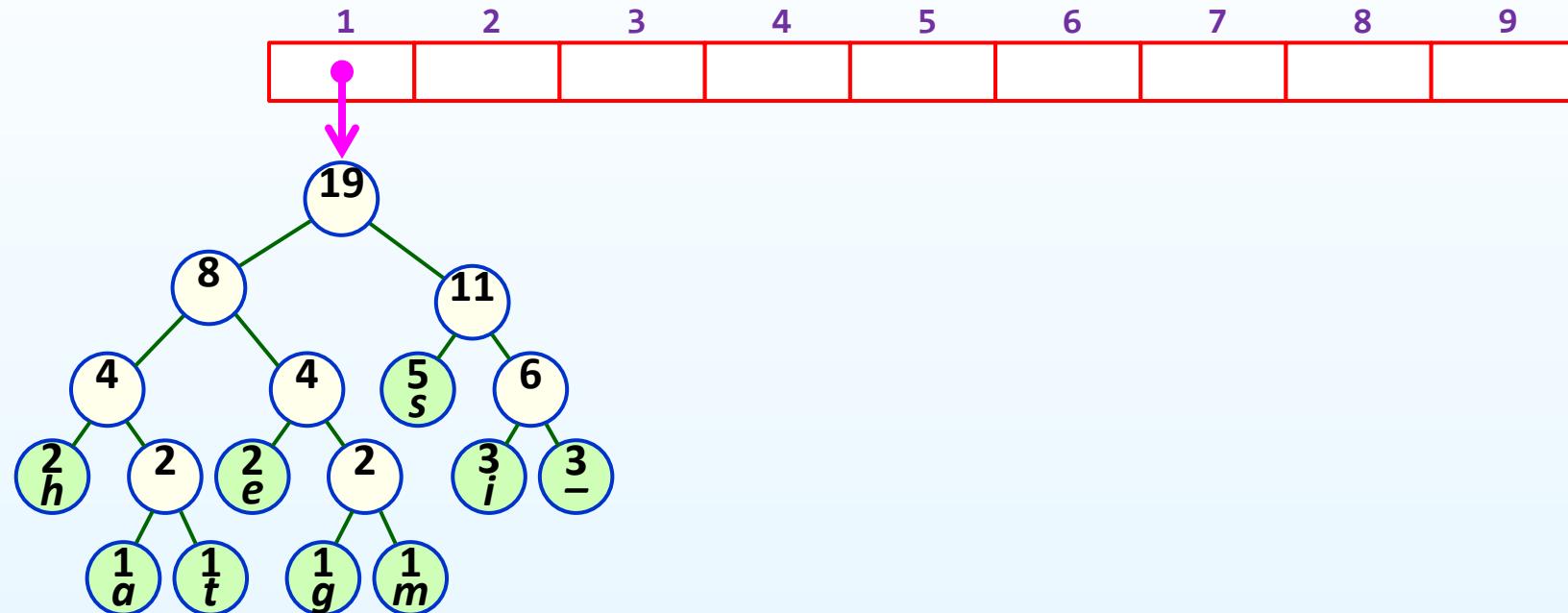
Example: Round 7

- Perform the `delete_min()` function 2 times
- Perform the `insert()` function



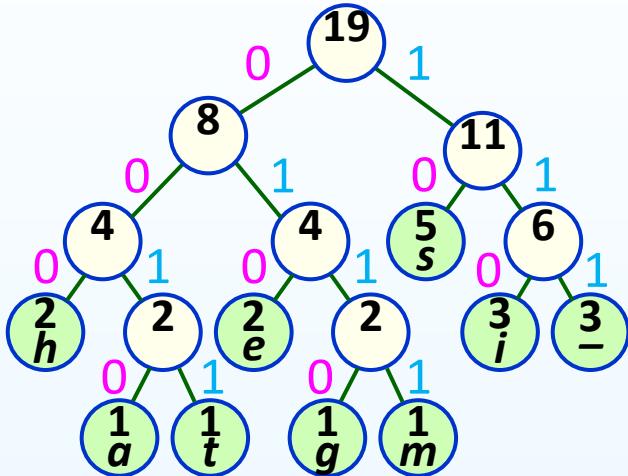
Example: Round 8

- Perform the `delete_min()` function 2 times
- Perform the `insert()` function



Example: Coding Table

- Finally, label edges from the node to its left and right children



Symbol	Frequency	Code
a	1	0010
e	2	010
g	1	0110
h	2	000
i	3	110
m	1	0111
s	5	10
t	1	0011
_	3	111

decoded message: *this is his message*



encoded message: 00110001101011111101011110001101011110111010101000100110010

Any Question?