Data Structures & Algorithms

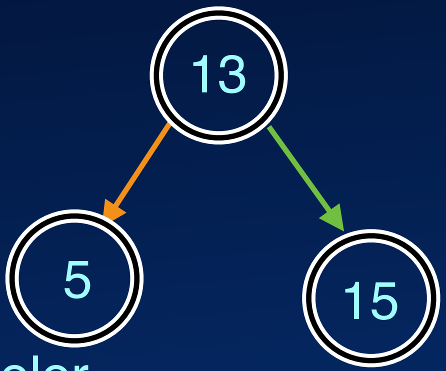
Subodh Kumar

(subodh@iitd.ac.in, Bharti 422)

Dept of Computer Sc. & Engg.

Red-Black Tree

Remove 15

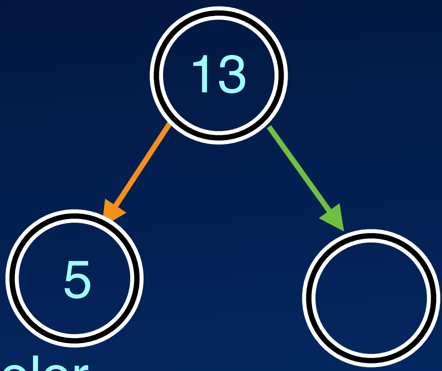


No red child: Recolor

a) There is no red coloring
b) 13 will have red color after 1 recoloring
c) 5 will have red color after 1 recoloring
d) Both 5 and 13 will have red color after 1 coloring each
e) 13 will have black color after 2 recolorings
f) 5 will have black color after 2 recolorings

Red-Black Tree

Remove 15



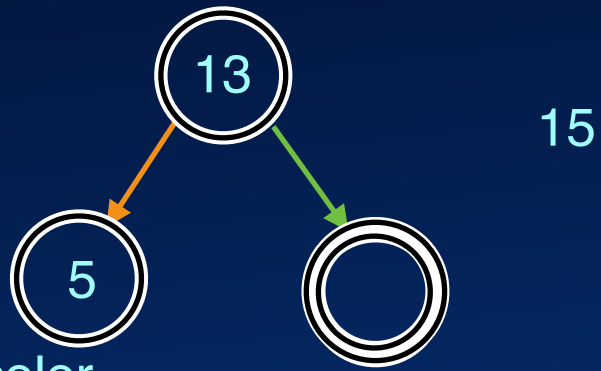
15

No red child: Recolor

a) There is no red coloring
b) 13 will have red color after 1 recoloring
c) 5 will have red color after 1 recoloring
d) Both 5 and 13 will have red color after 1 coloring each
e) 13 will have black color after 2 recolorings
f) 5 will have black color after 2 recolorings

Red-Black Tree

Remove 15

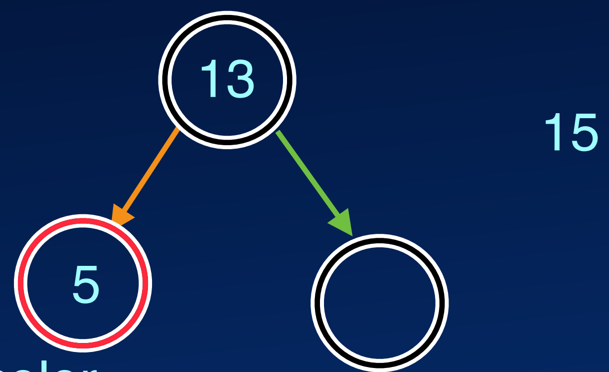


No red child: Recolor

a) There is no red coloring
b) 13 will have red color after 1 recoloring
c) 5 will have red color after 1 recoloring
d) Both 5 and 13 will have red color after 1 coloring each
e) 13 will have black color after 2 recolorings
f) 5 will have black color after 2 recolorings

Red-Black Tree

Remove 15

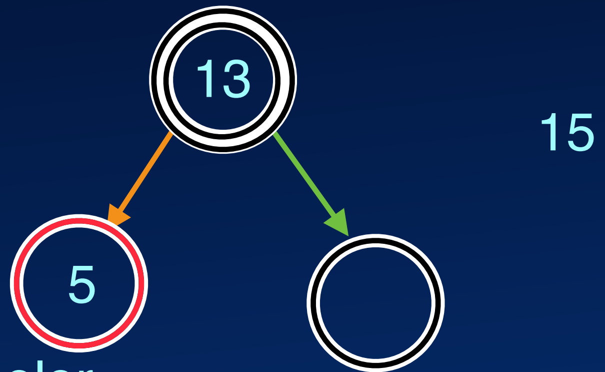


No red child: Recolor

a) There is no red coloring
b) 13 will have red color after 1 recoloring
c) 5 will have red color after 1 recoloring
d) Both 5 and 13 will have red color after 1 coloring each
e) 13 will have black color after 2 recolorings
f) 5 will have black color after 2 recolorings

Red-Black Tree

Remove 15

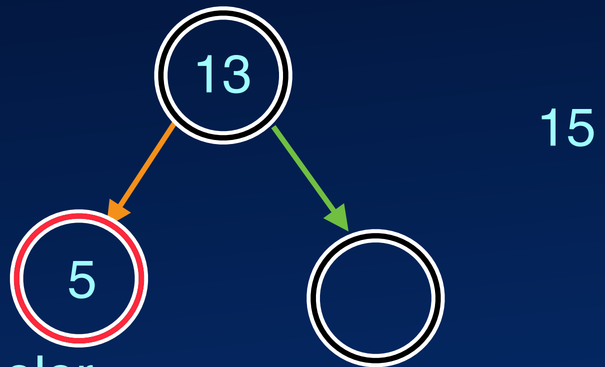


No red child: Recolor

a) There is no red coloring
b) 13 will have red color after 1 recoloring
c) 5 will have red color after 1 recoloring
d) Both 5 and 13 will have red color after 1 coloring each
e) 13 will have black color after 2 recolorings
f) 5 will have black color after 2 recolorings

Red-Black Tree

Remove 15

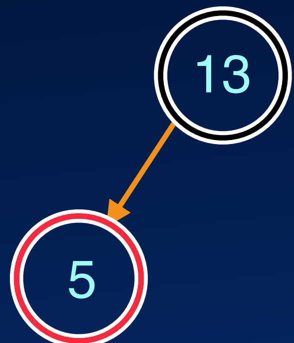


No red child: Recolor

a) There is no red coloring
b) 13 will have red color after 1 recoloring
c) 5 will have red color after 1 recoloring
d) Both 5 and 13 will have red color after 1 coloring each
e) 13 will have black color after 2 recolorings
f) 5 will have black color after 2 recolorings

Red-Black Tree

Remove 15



No red child: Recolor

a) There is no red coloring
b) 13 will have red color after 1 recoloring
c) 5 will have red color after 1 recoloring
d) Both 5 and 13 will have red color after 1 coloring each
e) 13 will have black color after 2 recolorings
f) 5 will have black color after 2 recolorings

Dictionaries

O(1) expected operations, no *next* traversal

Space efficient, no references needed

Hash table

Red-black

2-3 Tree

Dictionaries

O(1) expected operations, no *next* traversal

Space efficient, no references needed

Hash table

O(*length*) operation, *next* key possible in lexicographic order

Can be space-efficient if many common prefixes

Trie

Red-black

2-3 Tree

Dictionaries

O(1) expected operations, no *next* traversal

Space efficient, no references needed

Hash table

O(*length*) operation, *next* key possible in lexicographic order

Can be space-efficient if many common prefixes

Trie

O(log *n*) expected operation, *next* key in O(1)

Slightly more references than binary search tree

Skip List

Red-black

2-3 Tree

Dictionaries

O(1) expected operations, no *next* traversal

Space efficient, no references needed

Hash table

O(*length*) operation, *next* key possible in lexicographic order

Can be space-efficient if many common prefixes

Trie

O(log n) expected operation, *next* key in O(1)

Skip List

1 to 1.44 log n in height

Up to O(log n) restructures for delete (but can reduces future restructures)

Good with skewed (sorted) input sequences

AVL tree

Red-black

2-3 Tree

Dictionaries

O(1) expected operations, no *next* traversal

Space efficient, no references needed

Hash table

O(*length*) operation, *next* key possible in lexicographic order

Can be space-efficient if many common prefixes

Trie

O(log n) expected operation, *next* key in O(1)

Skip List

1 to 1.44 log n in height

AVL tree

Up to O(log n) restructures for delete (but can reduces future restructures)

Good with skewed (sorted) input sequences

Up to 2 log n high

Red-Black Tree

But only one restructure per update

Good for occasional run of sorted keys

2-3 Tree

Dictionaries

O(1) expected operations, no *next* traversal

Space efficient, no references needed

Hash table

O(*length*) operation, *next* key possible in lexicographic order

Can be space-efficient if many common prefixes

Trie

O(log n) expected operation, *next* key in O(1)

Skip List

1 to 1.44 log n in height

AVL tree

Up to O(log n) restructures for delete (but can reduces future restructures)

Good with skewed (sorted) input sequences

Up to 2 log n high

Red-Black Tree

But only one restructure per update

More work per node, but nodes can be cache-friendly

Works best for out of memory data structure

2-3 Tree

extended to a-b tree, Or just b-tree

Dictionaries

Unbalanced BSTs also OK

for random order of updated keys

O(*length*) operation, *next* key possible in lexicographic order

Can be space-efficient if many common prefixes

Trie

O(log n) expected operation, *next* key in O(1)

Skip List

1 to 1.44 log n in height

AVL tree

Up to O(log n) restructures for delete (but can reduces future restructures)

Good with skewed (sorted) input sequences

Up to 2 log n high

Red-Black Tree

But only one restructure per update

More work per node, but nodes can be cache-friendly

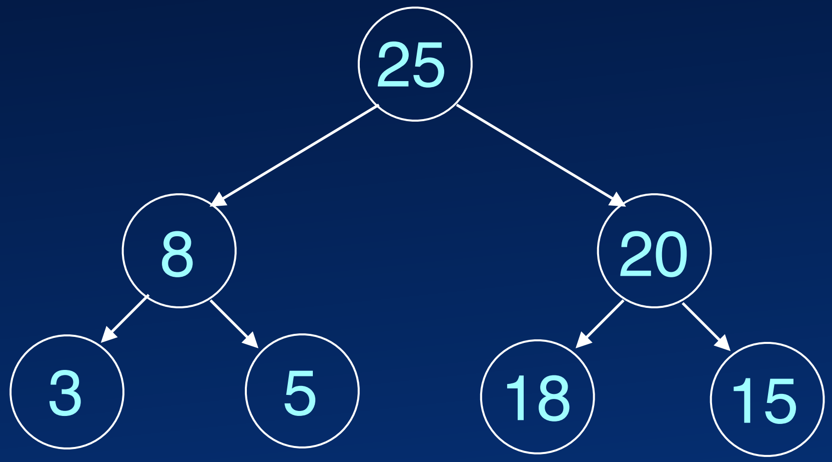
Works best for out of memory data structure

extended to a-b tree, Or just b-tree

2-3 Tree

Heap

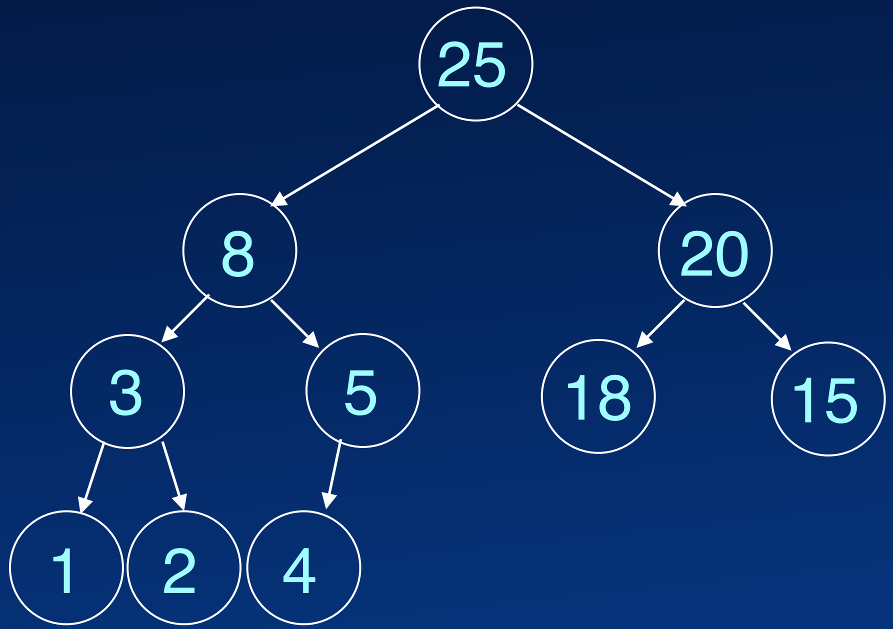
- Left-complete tree
- Comparable keys
- "Top" key in the root
  - For every subtree



Heap

Heap

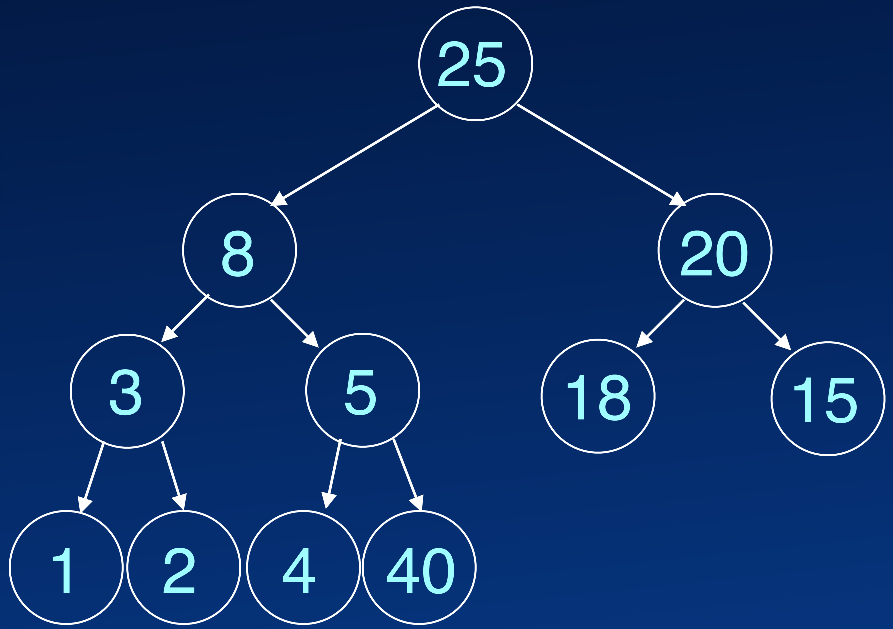
- Left-complete tree
- Comparable keys
- "Top" key in the root
  - For every subtree



Heap

Heap

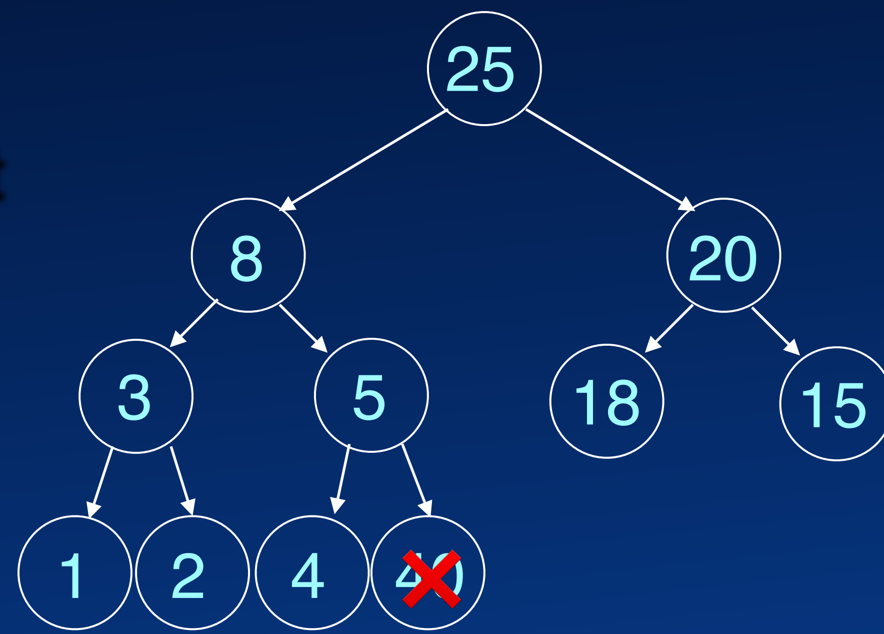
- Left-complete tree
- Comparable keys
- "Top" key in the root
  - For every subtree



Heap

Heap

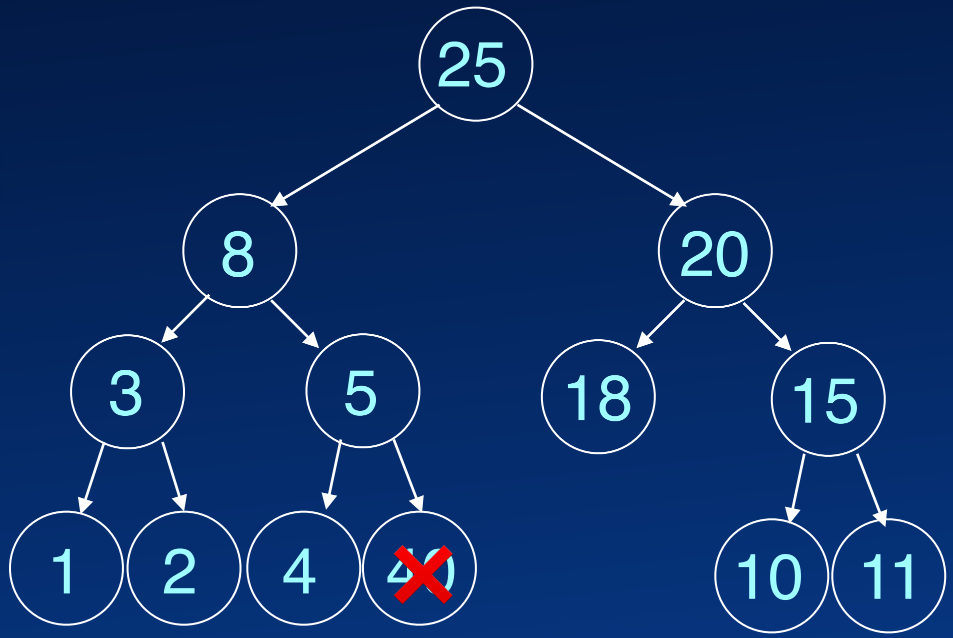
- Left-complete tree
- Comparable keys
- "Top" key in the root
    - For every subtree



Heap

Heap

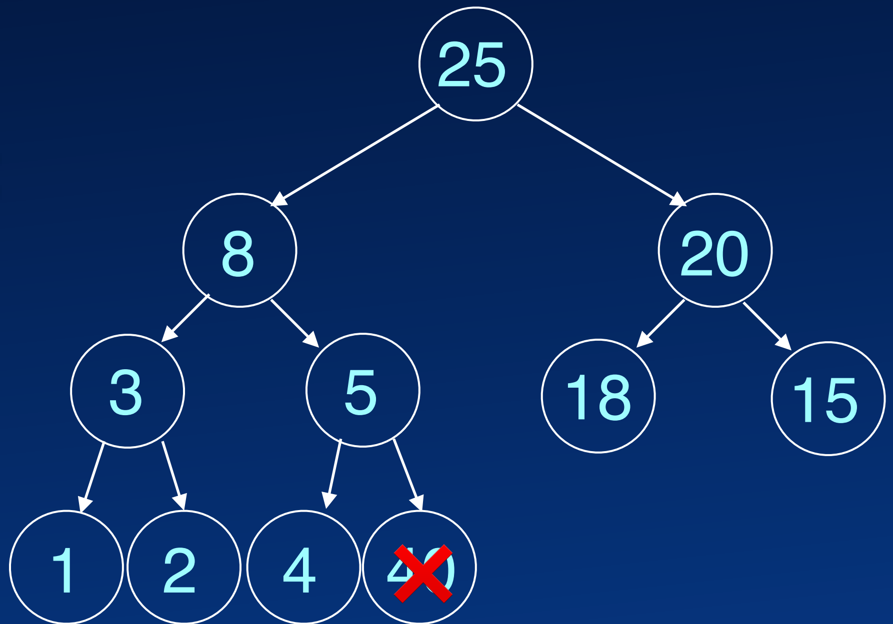
- Left-complete tree
- Comparable keys
- "Top" key in the root
  - For every subtree



Heap

Heap

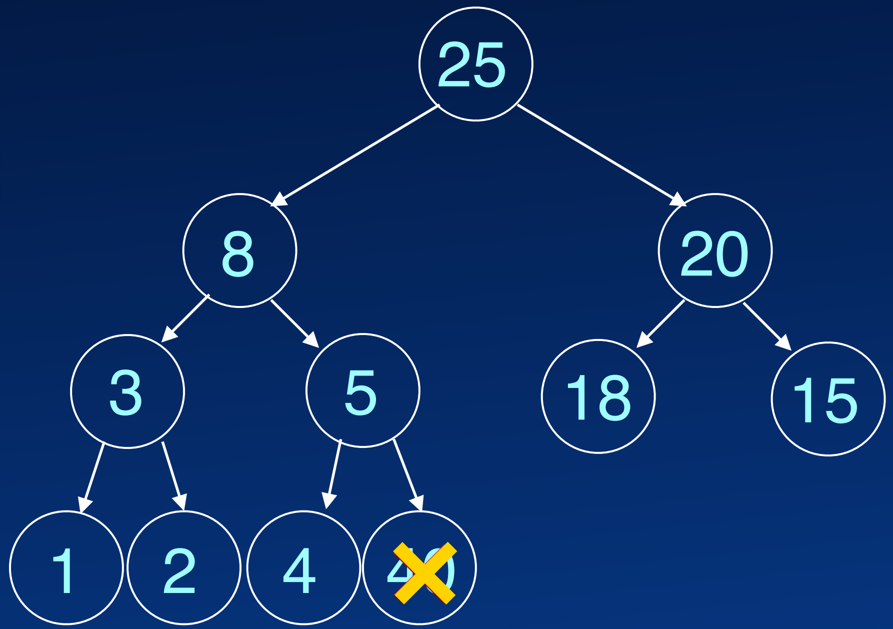
- Left-complete tree
- Comparable keys
- “Top” key in the root
  - For every subtree



Heap

Heap

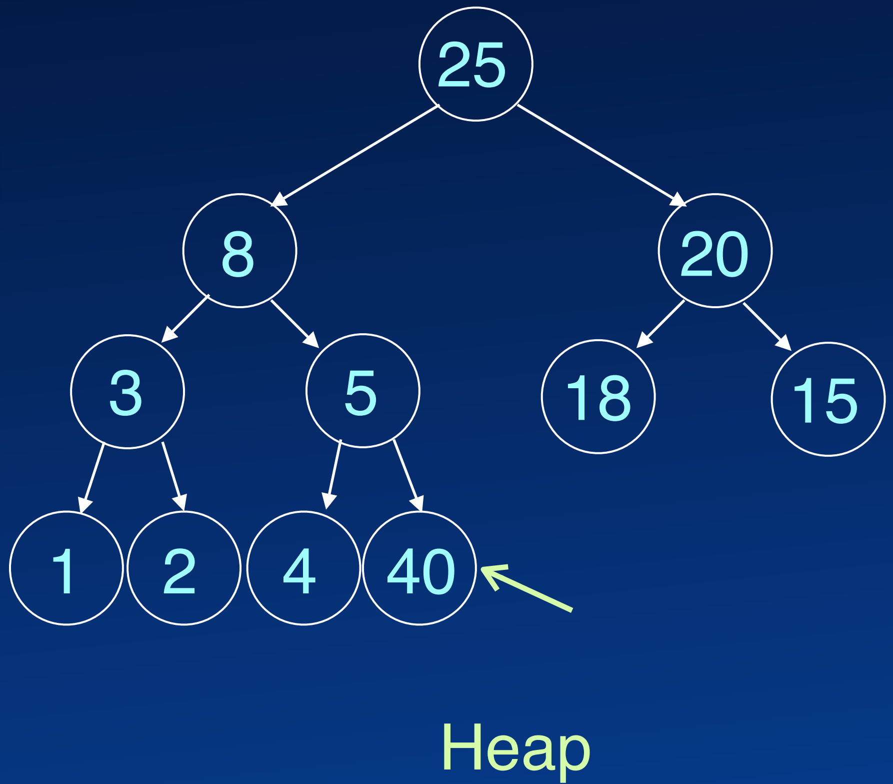
- Left-complete tree
- Comparable keys
- "Top" key in the root
  - For every subtree



Heap

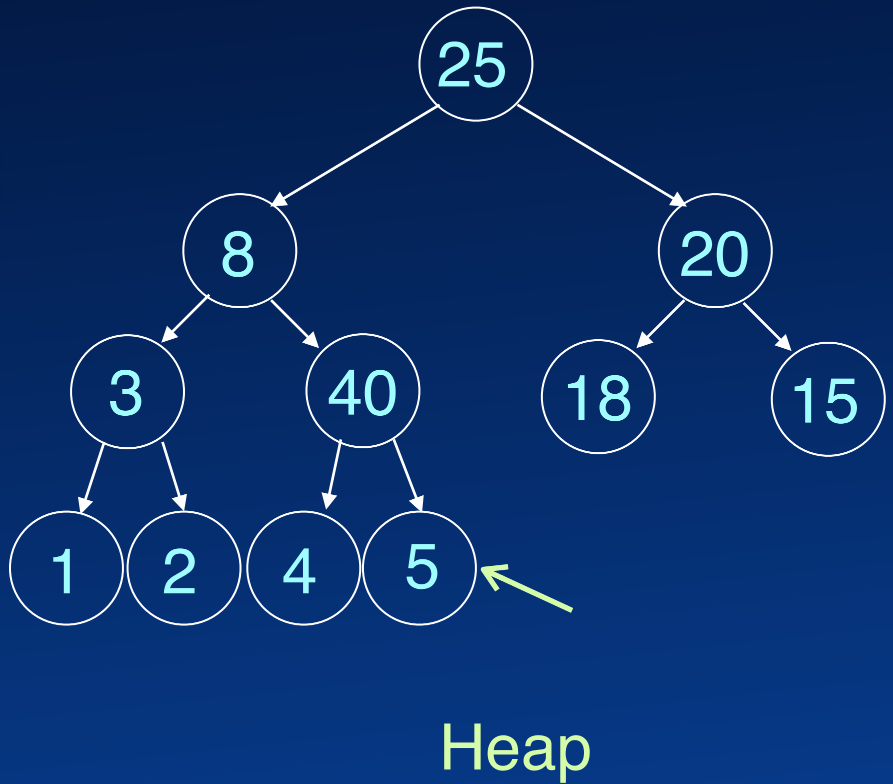
Heap

- Left-complete tree
- Comparable keys
- "Top" key in the root
  - For every subtree



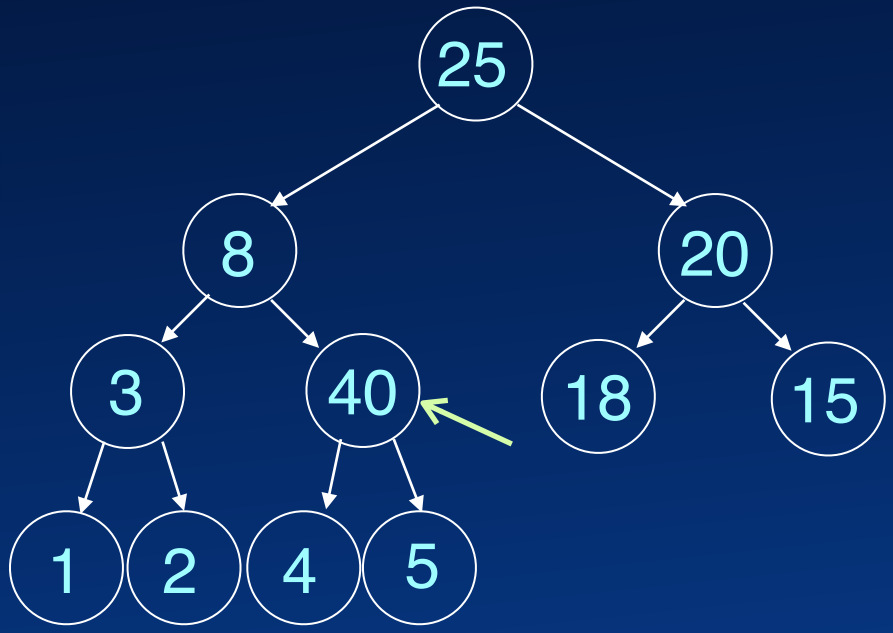
Heap

- Left-complete tree
- Comparable keys
- "Top" key in the root
  - For every subtree



Heap

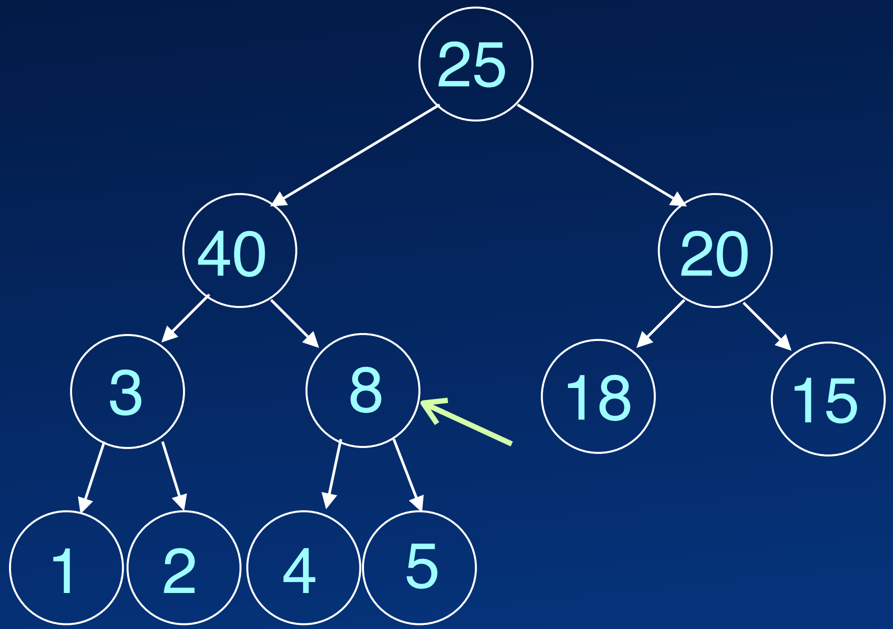
- Left-complete tree
- Comparable keys
- "Top" key in the root
  - For every subtree



Heap

Heap

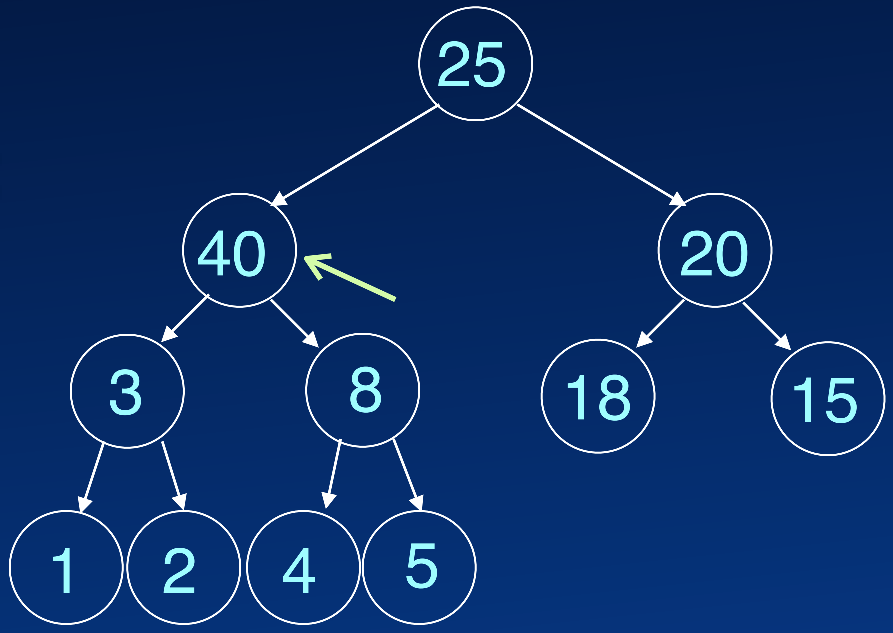
- Left-complete tree
- Comparable keys
- "Top" key in the root
  - For every subtree



Heap

Heap

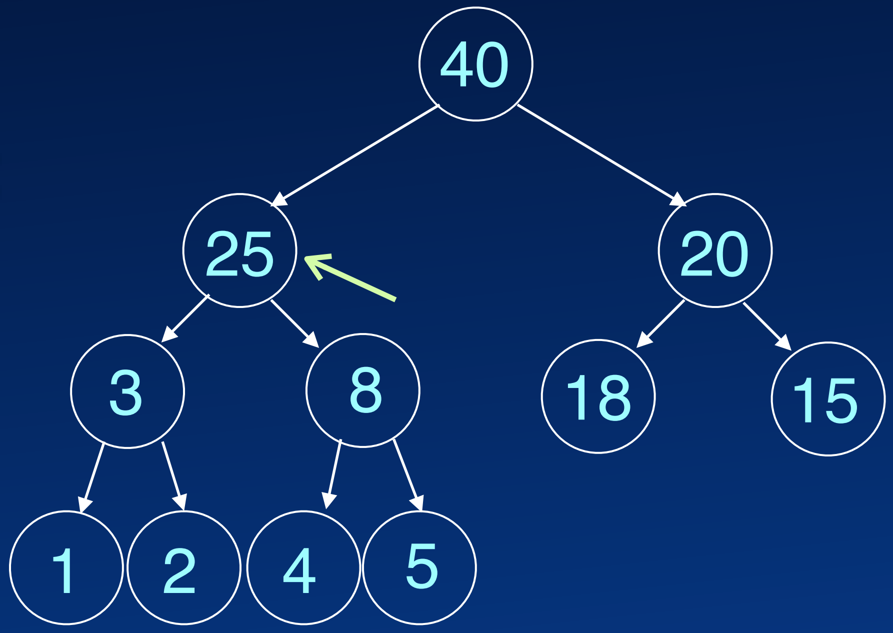
- Left-complete tree
- Comparable keys
- "Top" key in the root
  - For every subtree



Heap

Heap

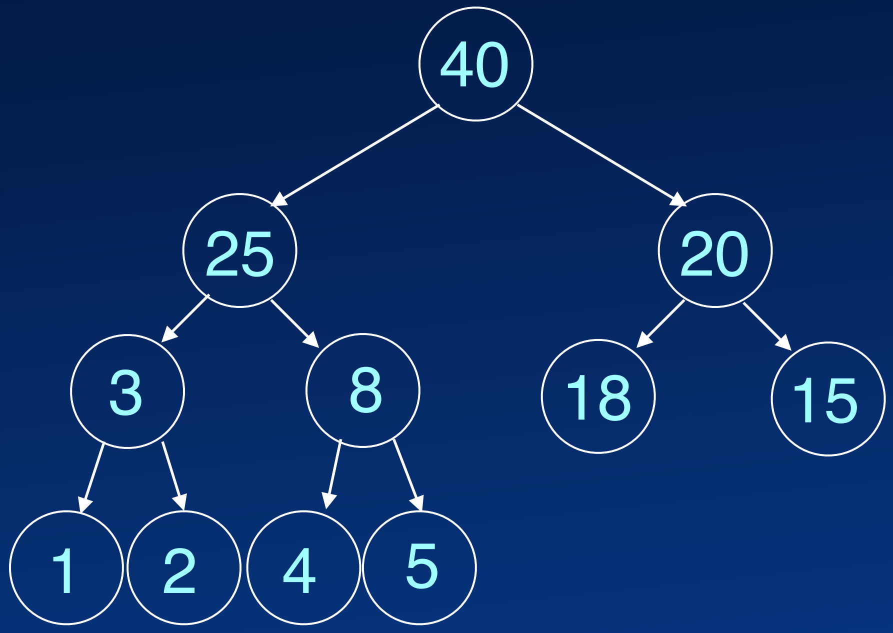
- Left-complete tree
- Comparable keys
- "Top" key in the root
  - For every subtree



Heap

Heap

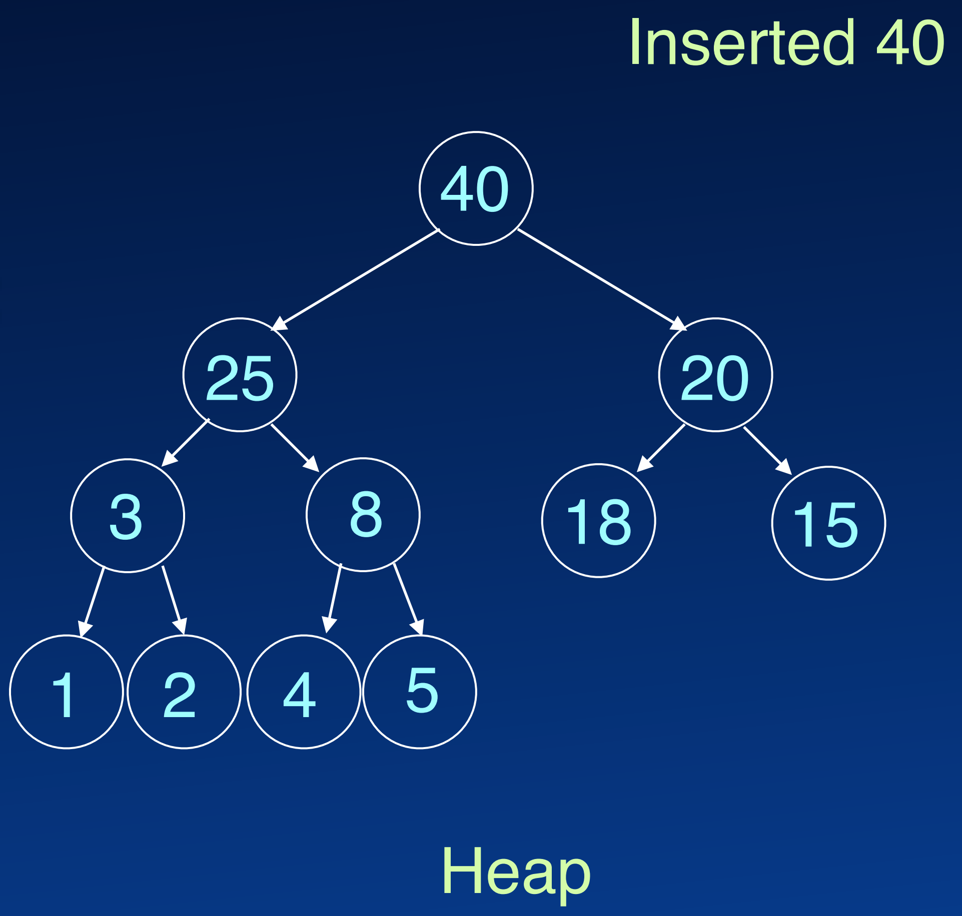
- Left-complete tree
- Comparable keys
- "Top" key in the root
  - For every subtree



Heap

Heap

- Left-complete tree
- Comparable keys
- "Top" key in the root
  - For every subtree

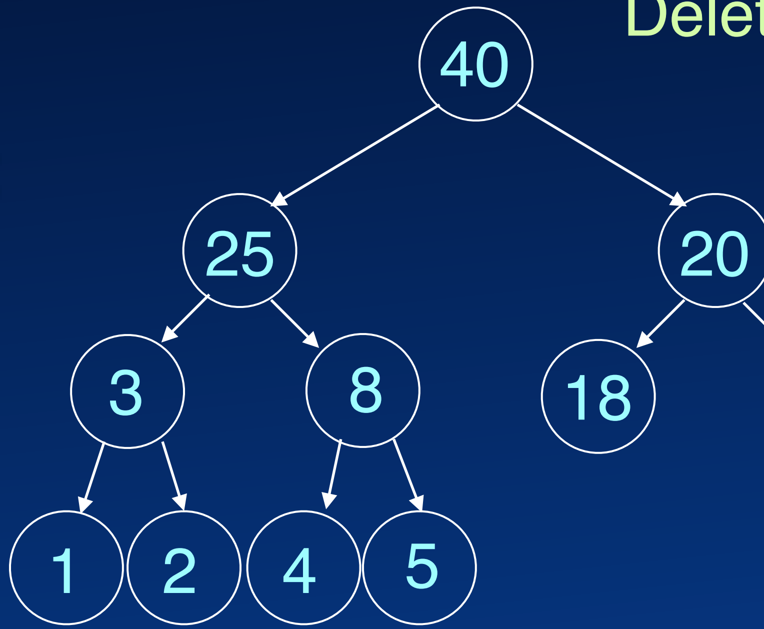


Heap

- Left-complete tree
- Comparable keys
- "Top" key in the root
  - For every subtree

Inserted 40

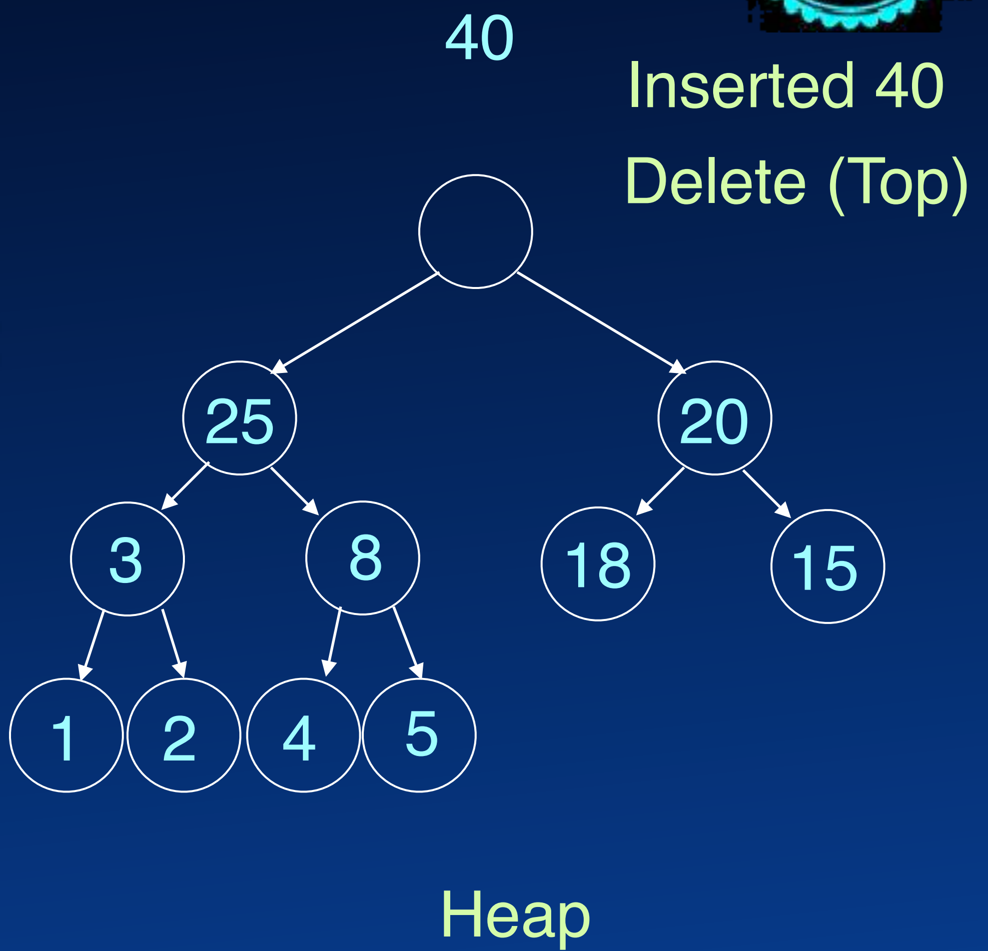
Delete (Top)



Heap

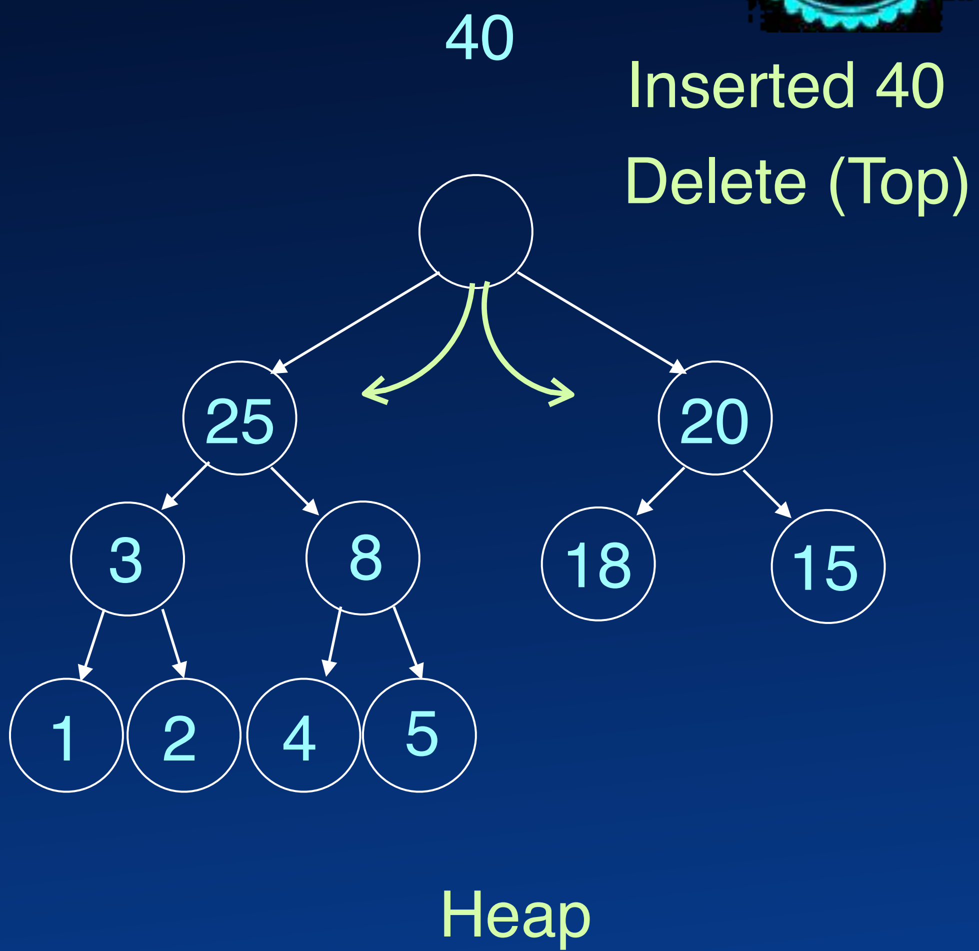
Heap

- Left-complete tree
- Comparable keys
- "Top" key in the root
  - For every subtree



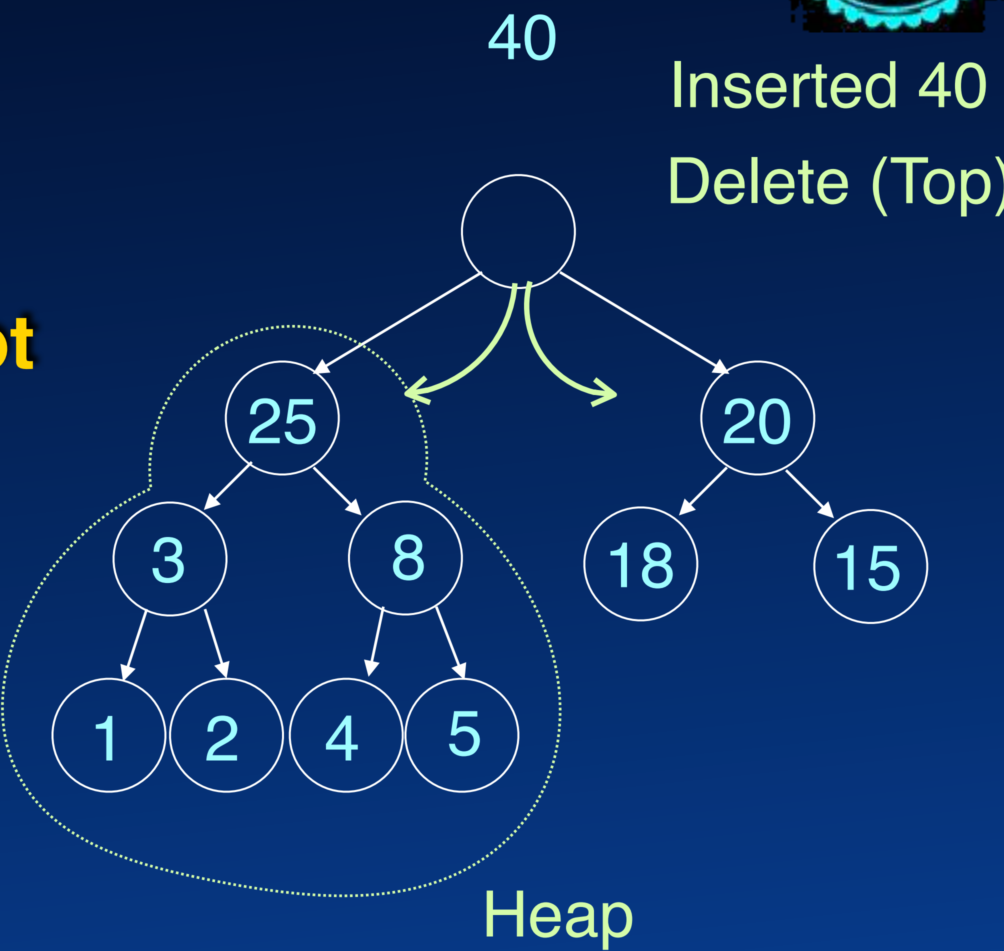
Heap

- **Left-complete tree**
- **Comparable keys**
- **“Top” key in the root**
  - **For every subtree**



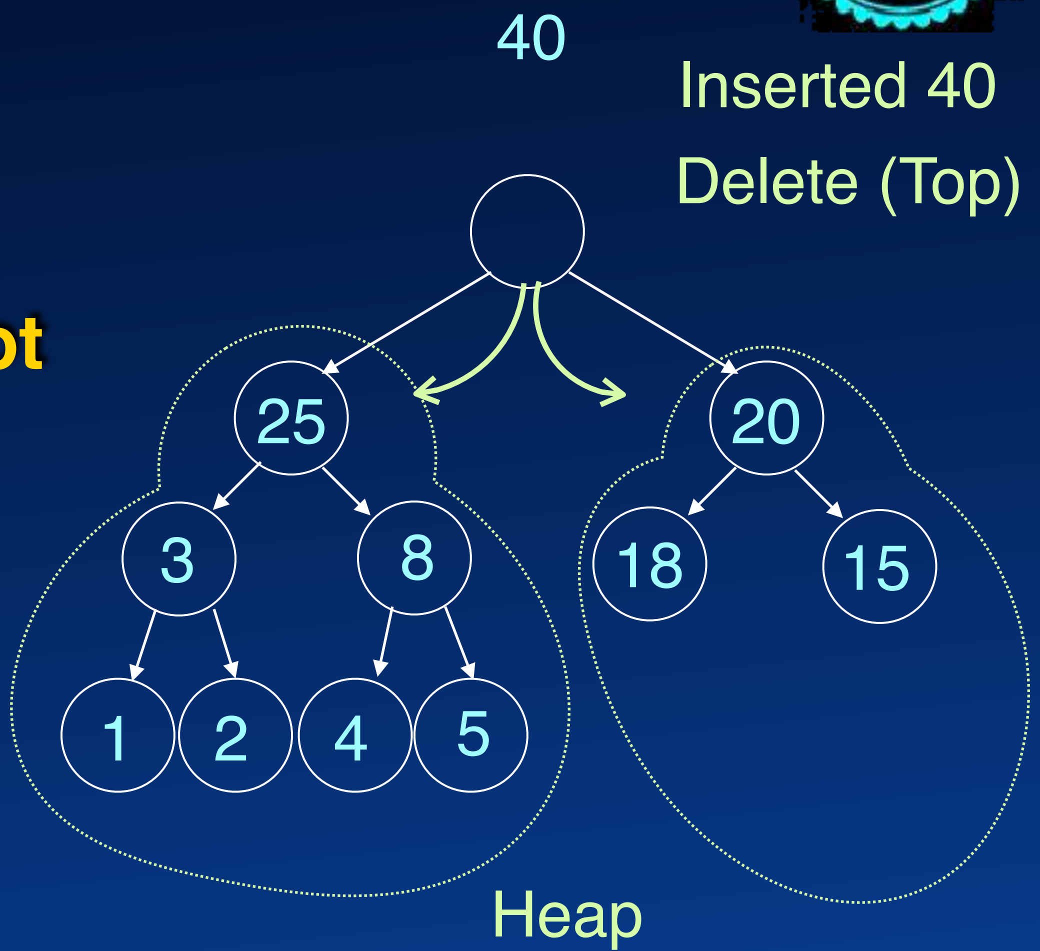
Heap

- **Left-complete tree**
- **Comparable keys**
- **“Top” key in the root**
  - **For every subtree**



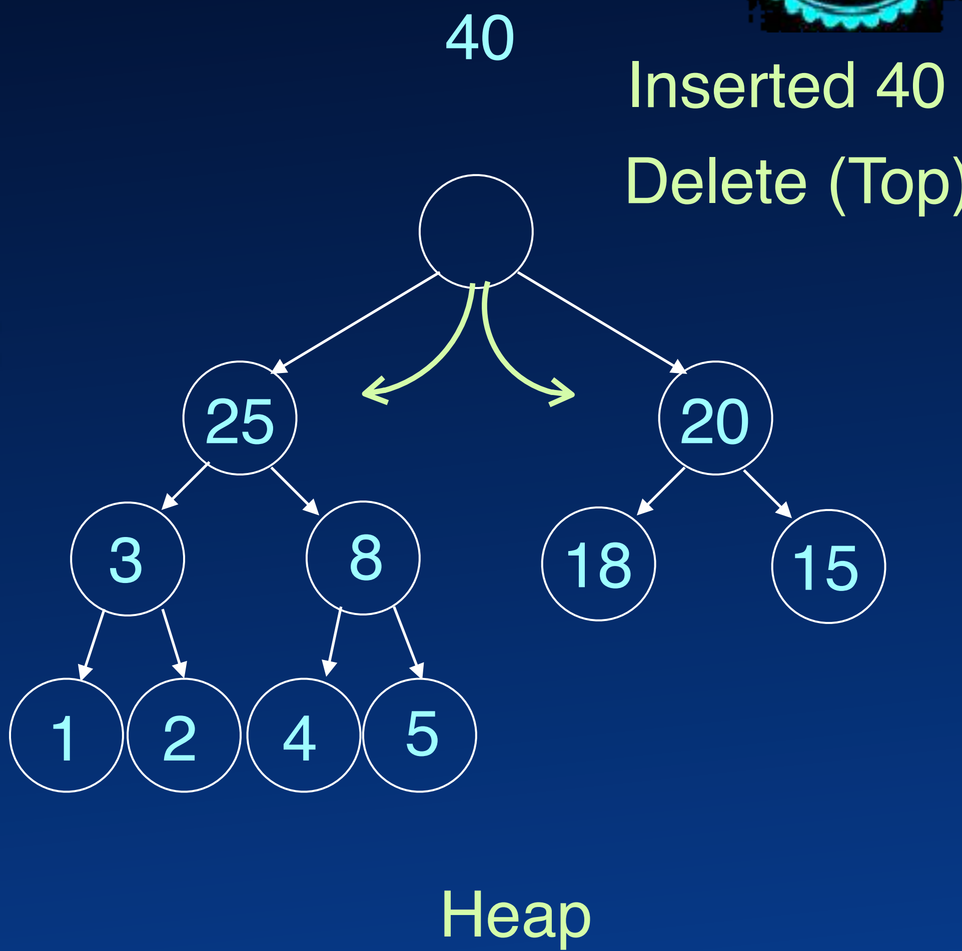
Heap

- **Left-complete tree**
- **Comparable keys**
- **“Top” key in the root**
  - **For every subtree**



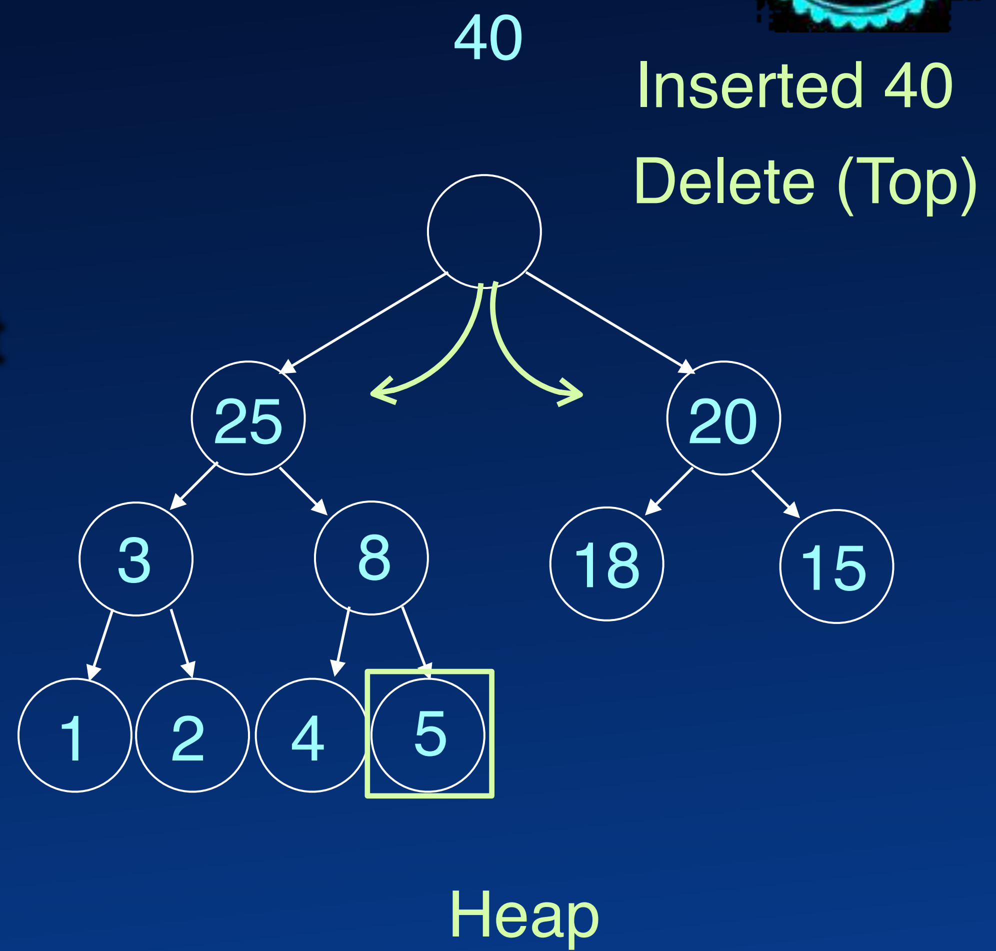
Heap

- **Left-complete tree**
- **Comparable keys**
- **"Top" key in the root**
  - **For every subtree**



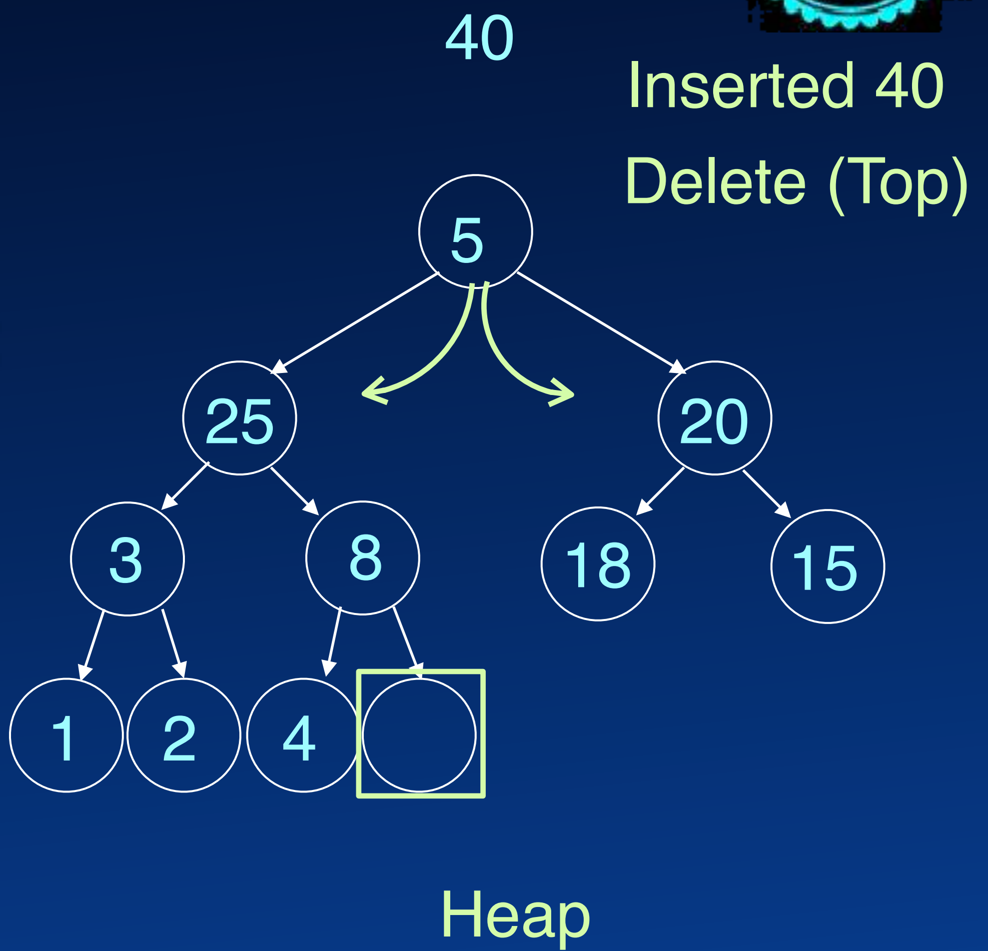
Heap

- Left-complete tree
- Comparable keys
- "Top" key in the root
  - For every subtree



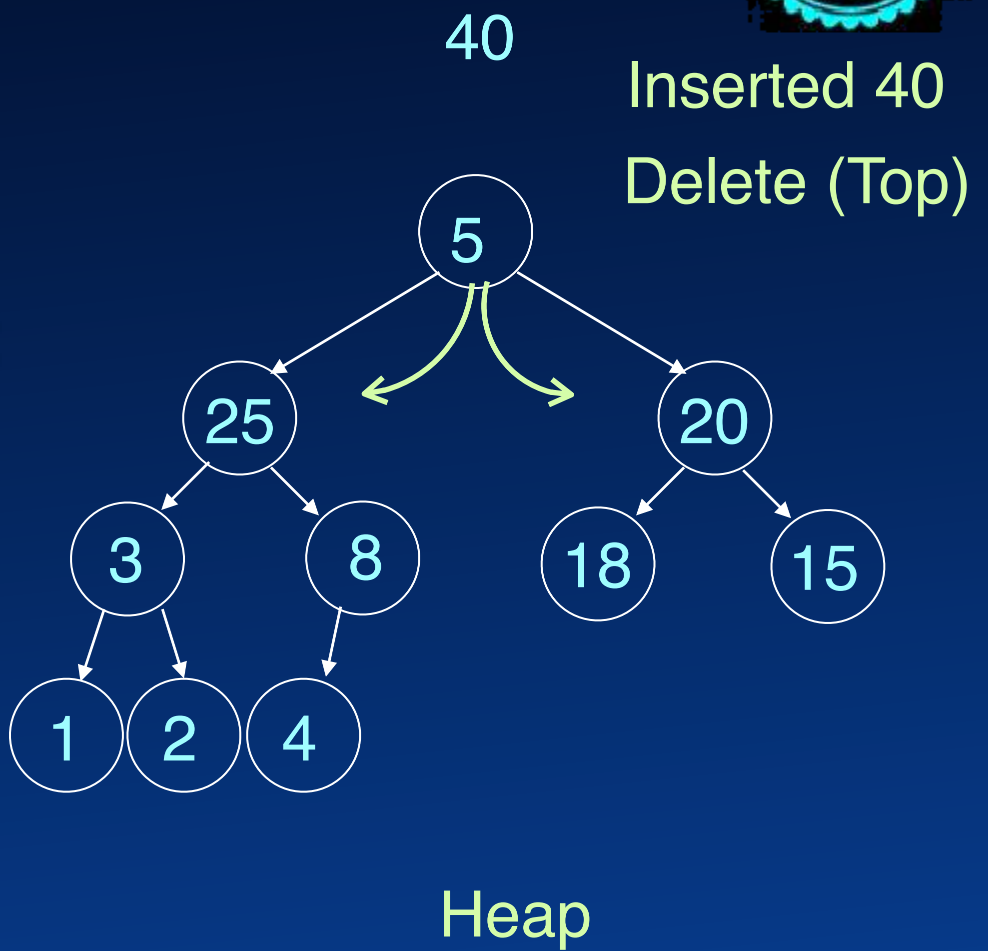
Heap

- Left-complete tree
- Comparable keys
- "Top" key in the root
  - For every subtree



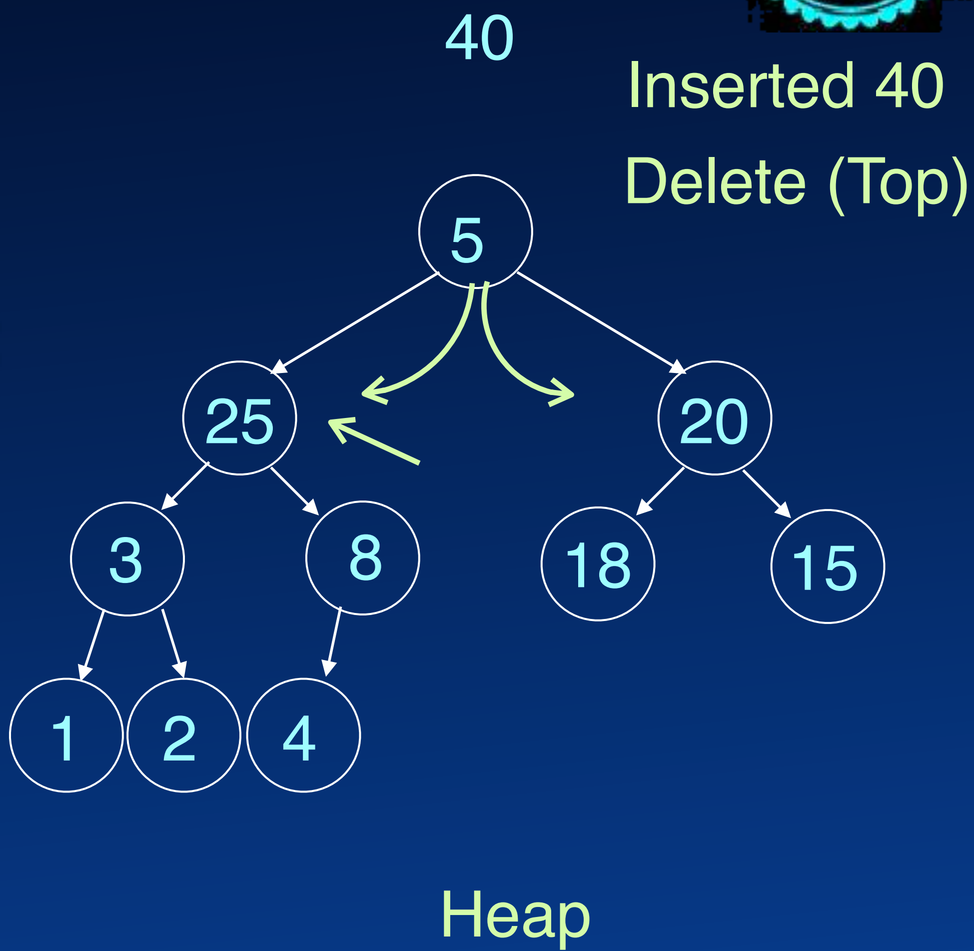
Heap

- **Left-complete tree**
- **Comparable keys**
- **"Top" key in the root**
  - **For every subtree**



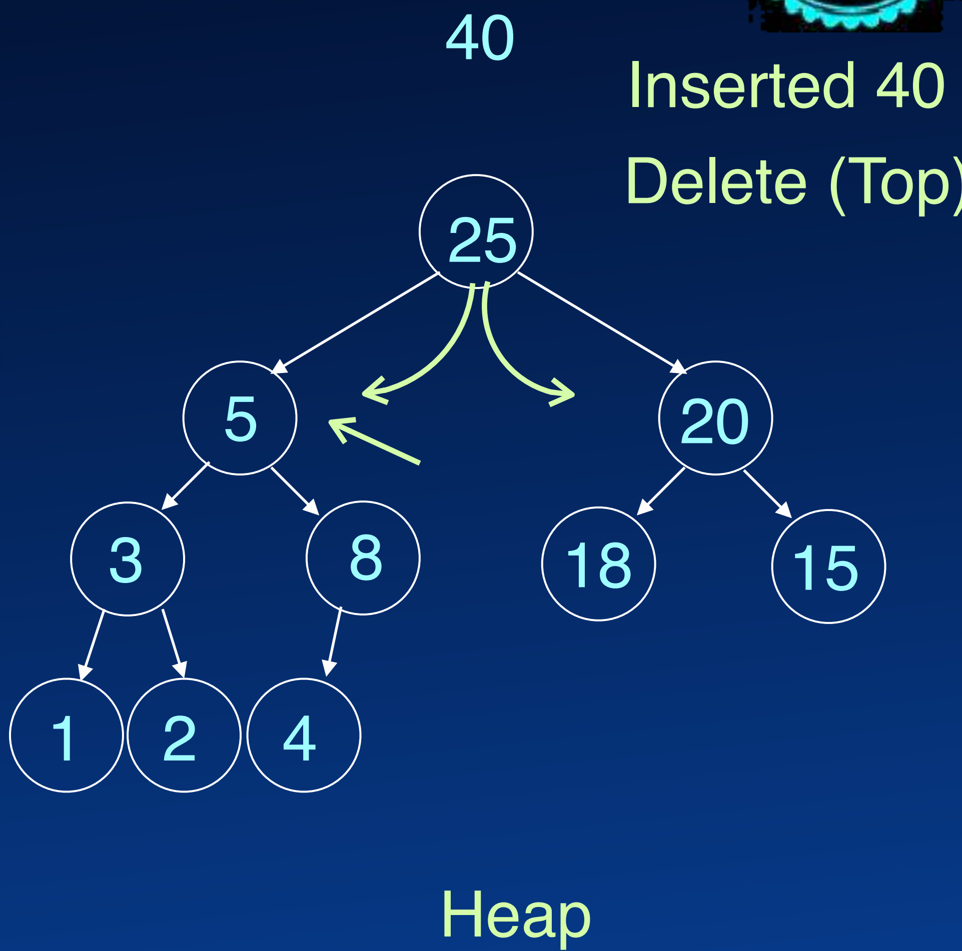
Heap

- **Left-complete tree**
- **Comparable keys**
- **"Top" key in the root**
  - **For every subtree**



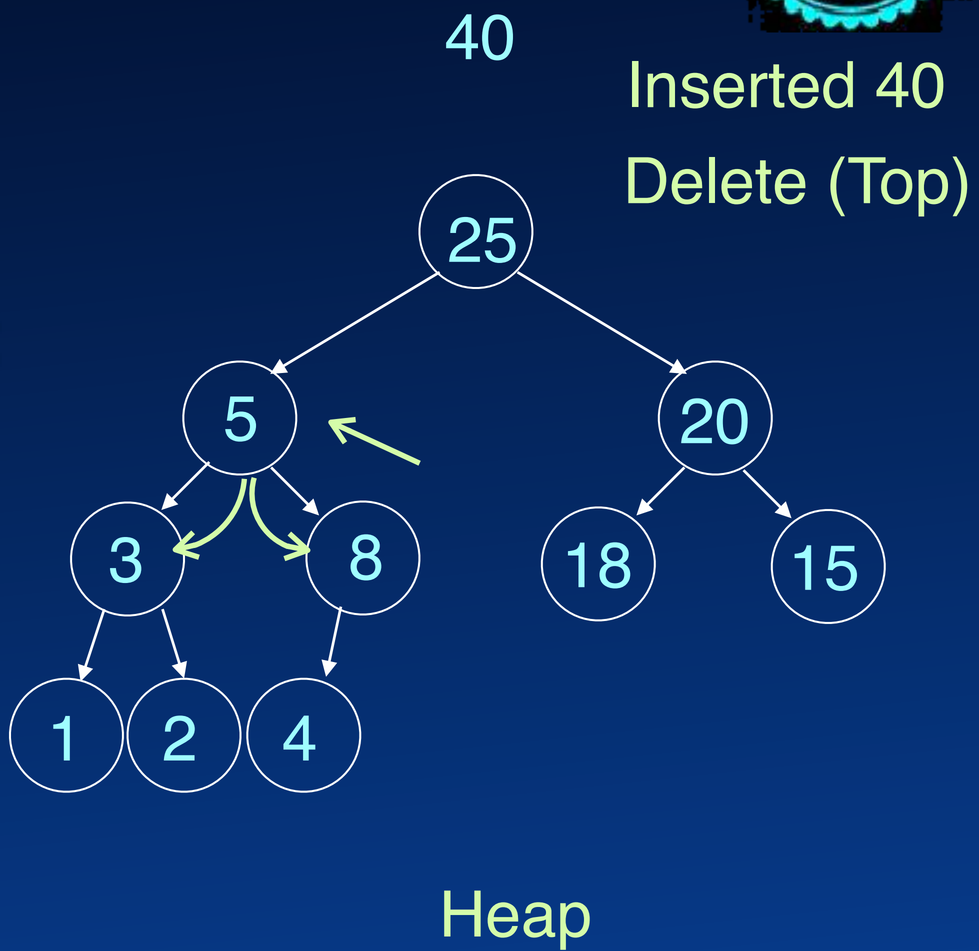
Heap

- **Left-complete tree**
- **Comparable keys**
- **“Top” key in the root**
  - **For every subtree**



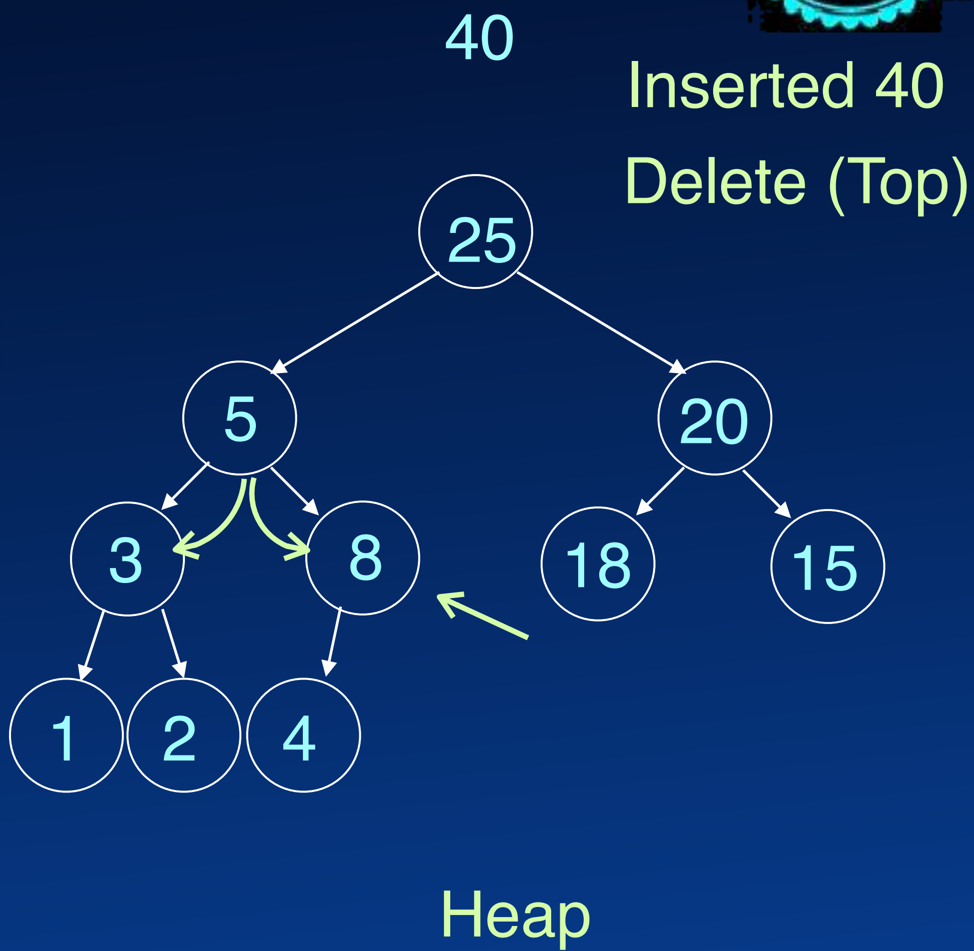
Heap

- Left-complete tree
- Comparable keys
- "Top" key in the root
  - For every subtree



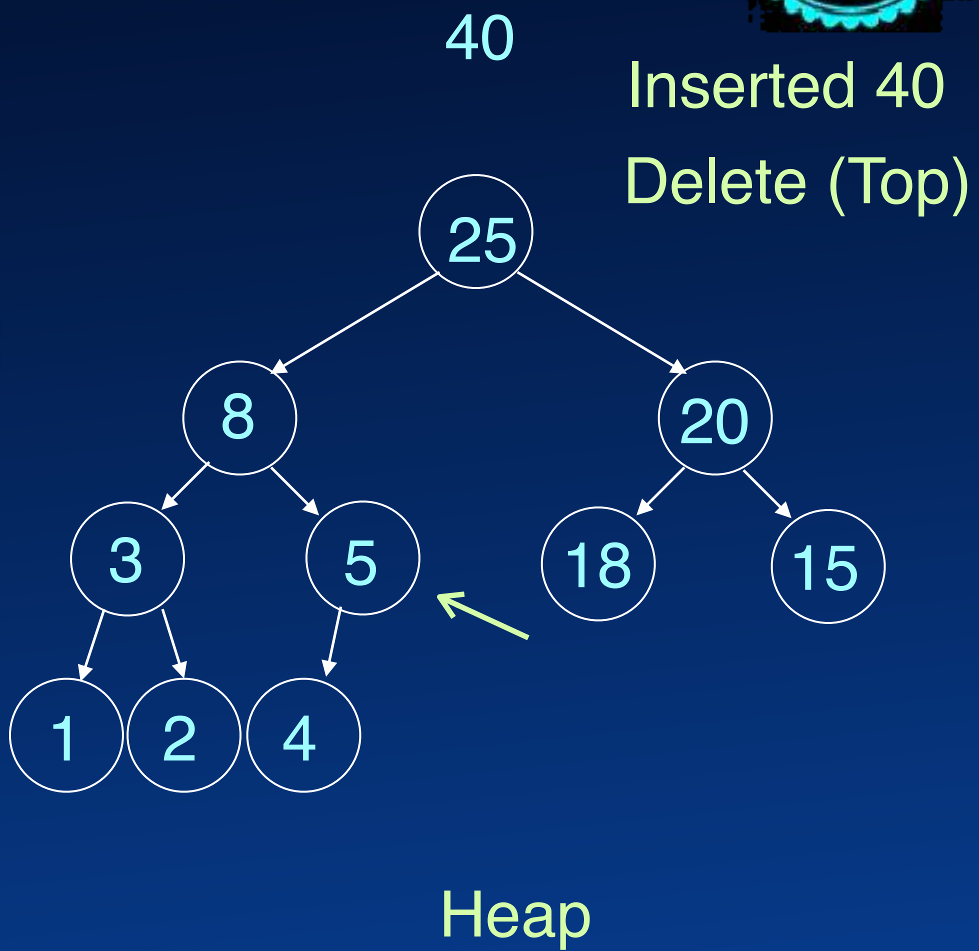
Heap

- Left-complete tree
- Comparable keys
- "Top" key in the root
  - For every subtree



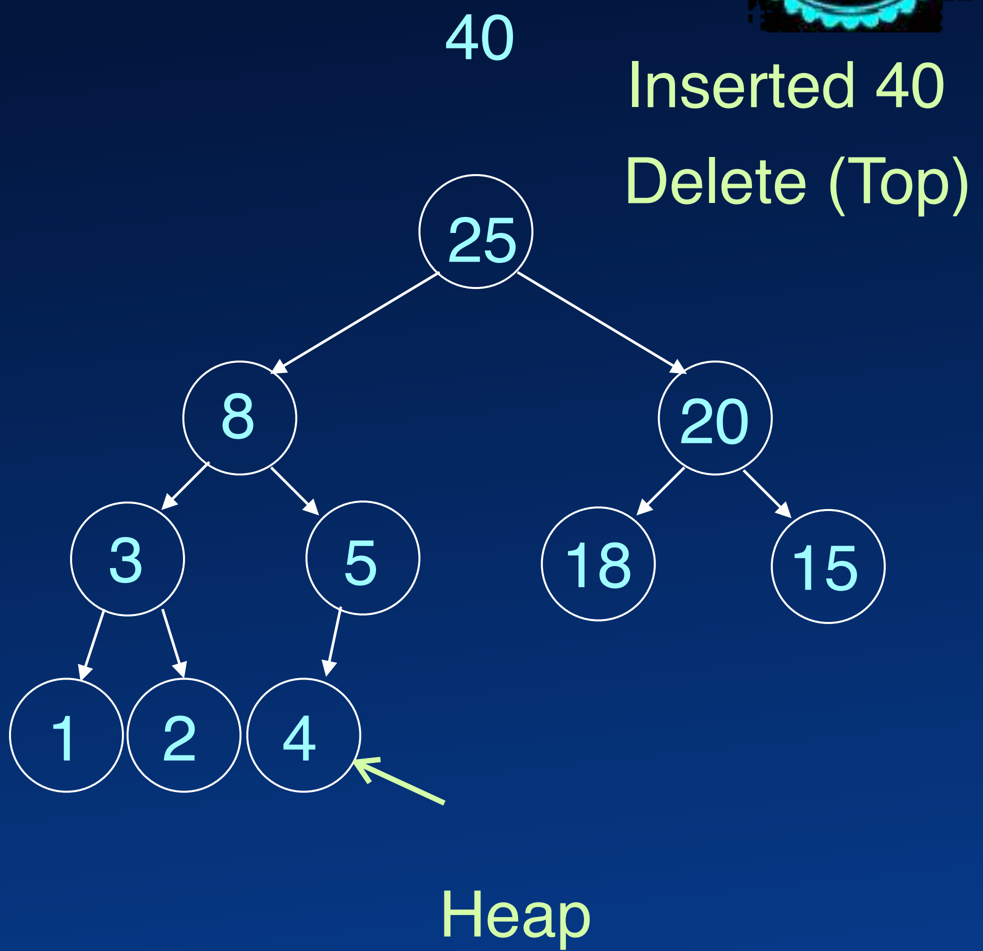
Heap

- **Left-complete tree**
- **Comparable keys**
- **"Top" key in the root**
  - **For every subtree**



Heap

- Left-complete tree
- Comparable keys
- "Top" key in the root
  - For every subtree



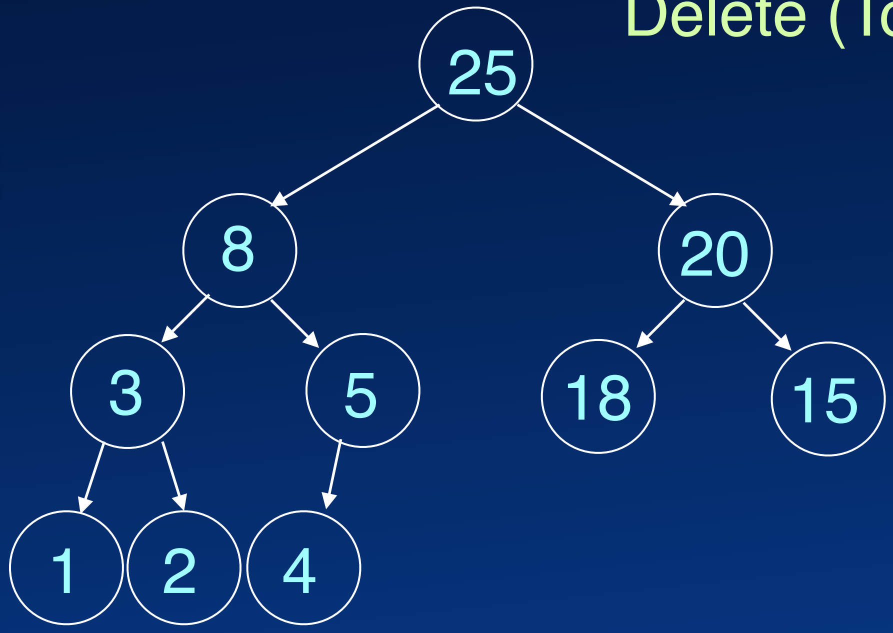
Heap

- **Left-complete tree**
- **Comparable keys**
- **“Top” key in the root**
  - **For every subtree**

40

Inserted 40

Delete (Top)



Heap

Heap

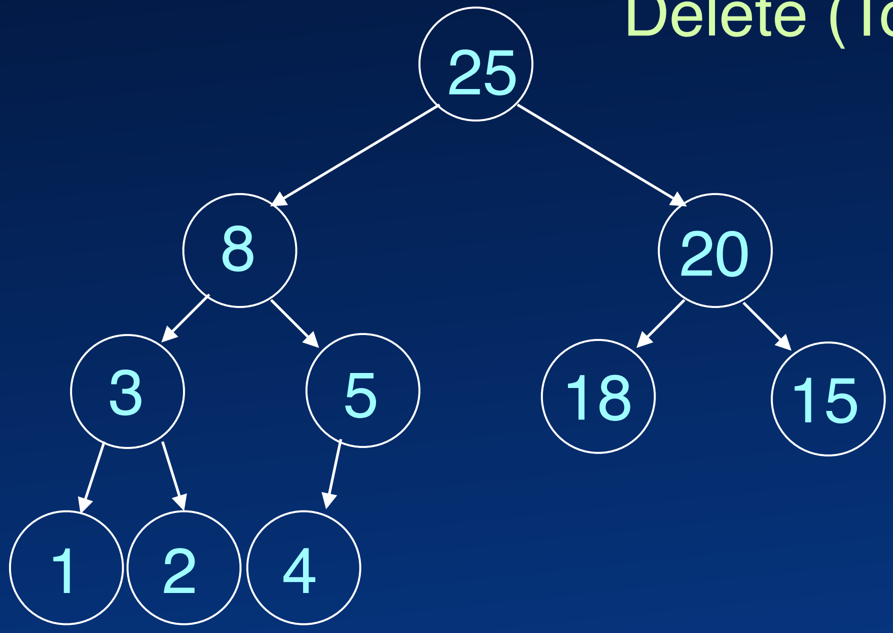
- **Left-complete tree**
- **Comparable keys**
- **"Top" key in the root**
  - **For every subtree**

Insert:
  Add node at next spot
  Bubble-up
Delete:
  Remove root
  Replace with last spot
  Bubble-down

40

Inserted 40

Delete (Top)



Heap

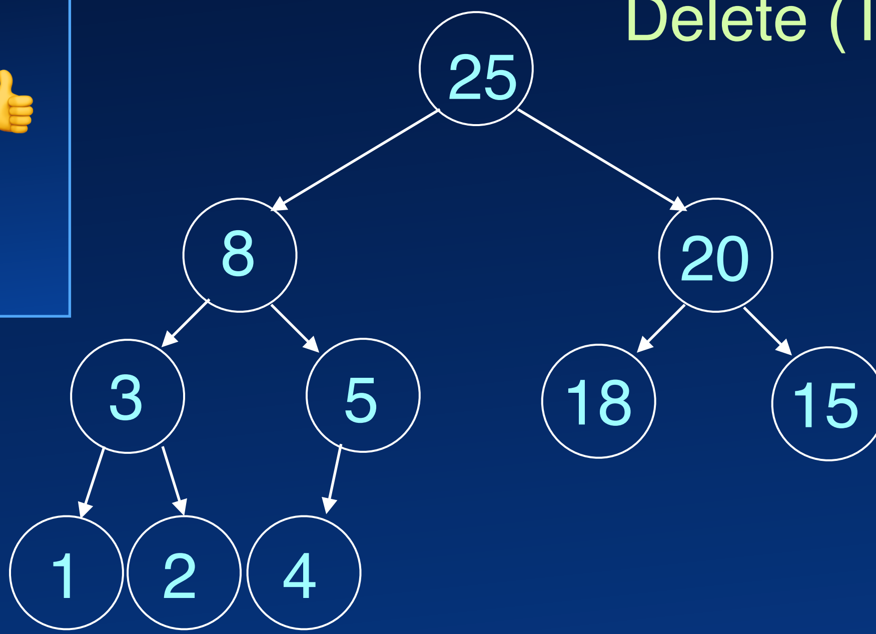
Heap

40

Inserted 40

Delete (Top)

Bubble up:
- if no parent 👍
- if(key lower-than parent.key) 👍
- swapwith(parent)
- parent.bubbleup()



Heap

Insert:
- Add node at next spot
- Bubble-up

Delete:
- Remove root
- Replace with last spot
- Bubble-down

Heap

Bubble down:

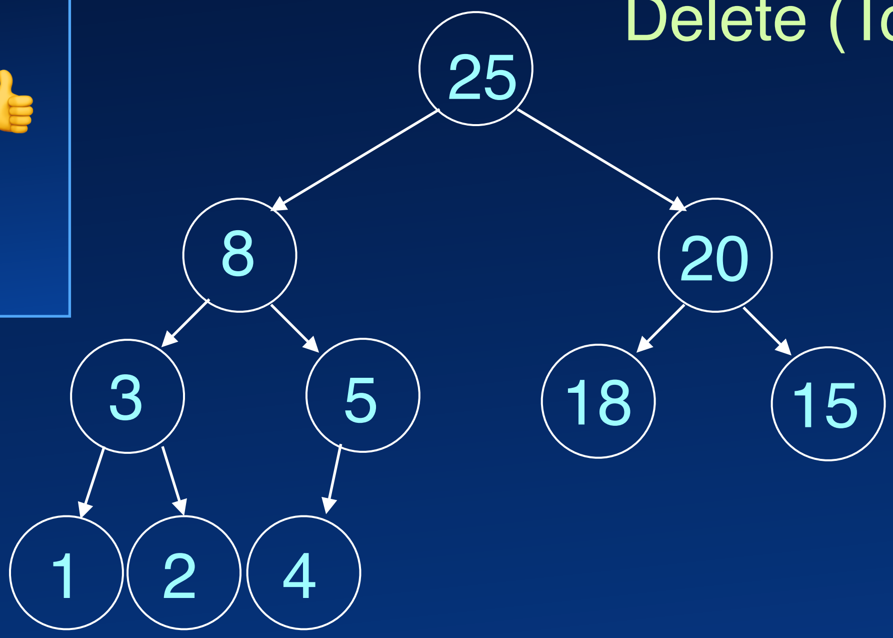
Bubble up:
  if no parent 👍
  if(key lower-than parent.key) 👍
  swapwith(parent)
  parent.bubbleup()

Insert:
  Add node at next spot
  Bubble-up
Delete:
  Remove root
  Replace with last spot
  Bubble-down

40

Inserted 40

Delete (Top)



Heap

Heap

Bubble down:

Bubble up:
- if no parent 👍
- if(key lower-than parent.key) 👍
- swapwith(parent)
- parent.bubbleup()

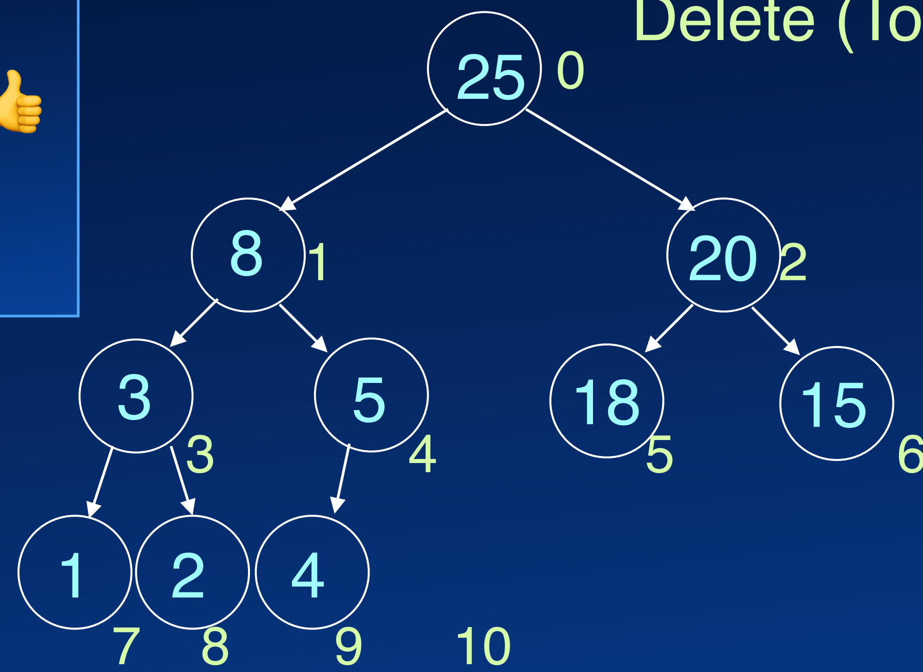
Insert:
- Add node at next spot
- Bubble-up

Delete:
- Remove root
- Replace with last spot
- Bubble-down

40

Inserted 40

Delete (Top)



Heap

Heap

Bubble down:

Bubble up:
- if no parent 👍
- if(key lower-than parent.key) 👍
- swapwith(parent)
- parent.bubbleup()

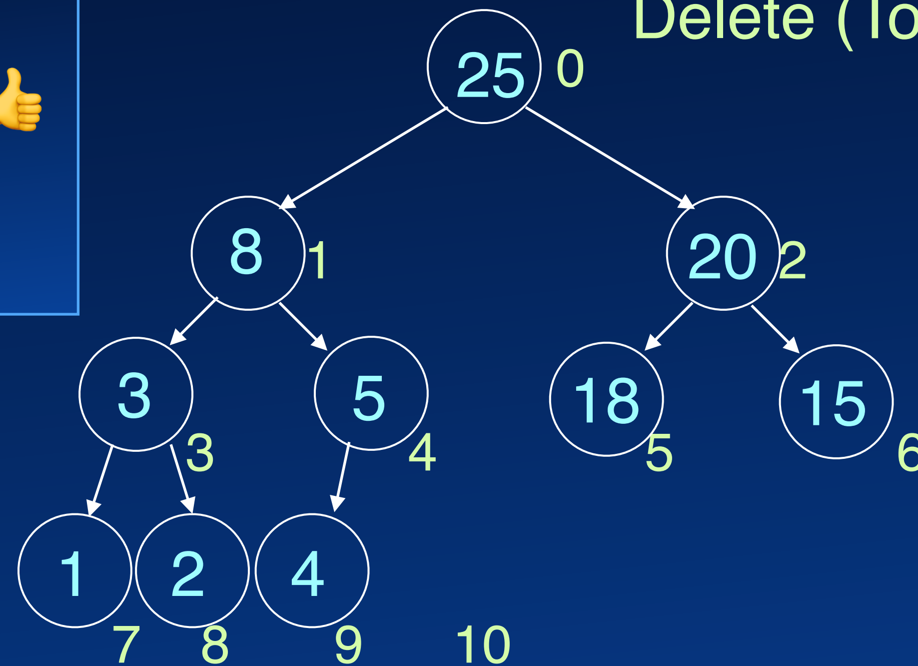
Insert:
- Add node at next spot
- Bubble-up

Delete:
- Remove root
- Replace with last spot
- Bubble-down

40

Inserted 40

Delete (Top)



Heap

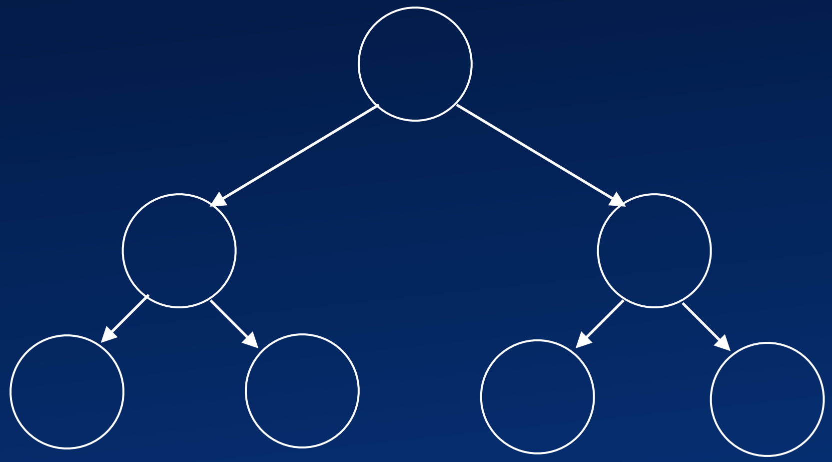
child-index = 2*parent-index + {1,2}

parent-index = (child-index-1)/2

In which node may the third largest element of a heap be?

mail: col106quiz@cse.iitd.ac.in
format: 1,2,3,4,5,6

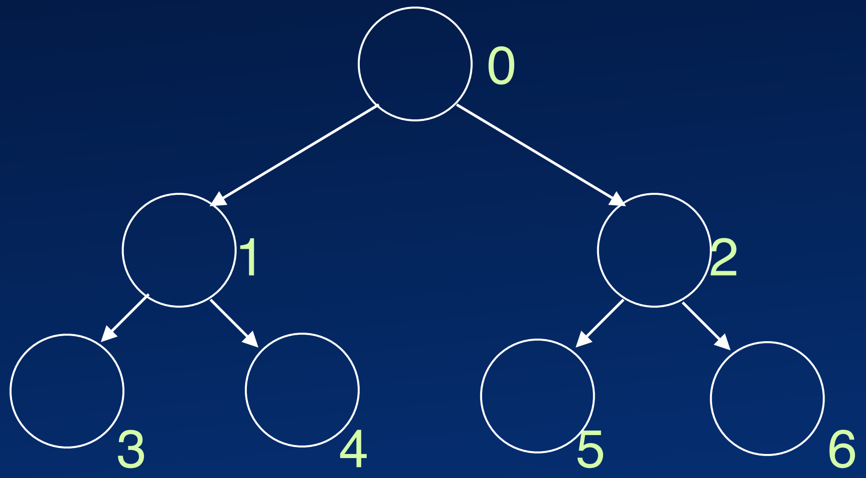
- 1
- 2
- 3
- 4
- 5
- 6



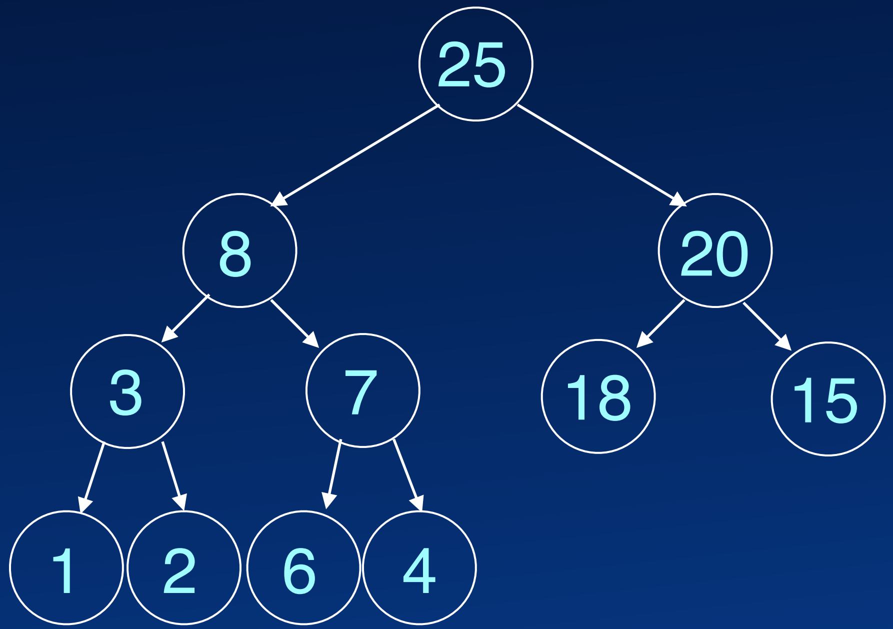
In which node may the third largest element of a heap be?

mail: col106quiz@cse.iitd.ac.in
format: 1,2,3,4,5,6

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

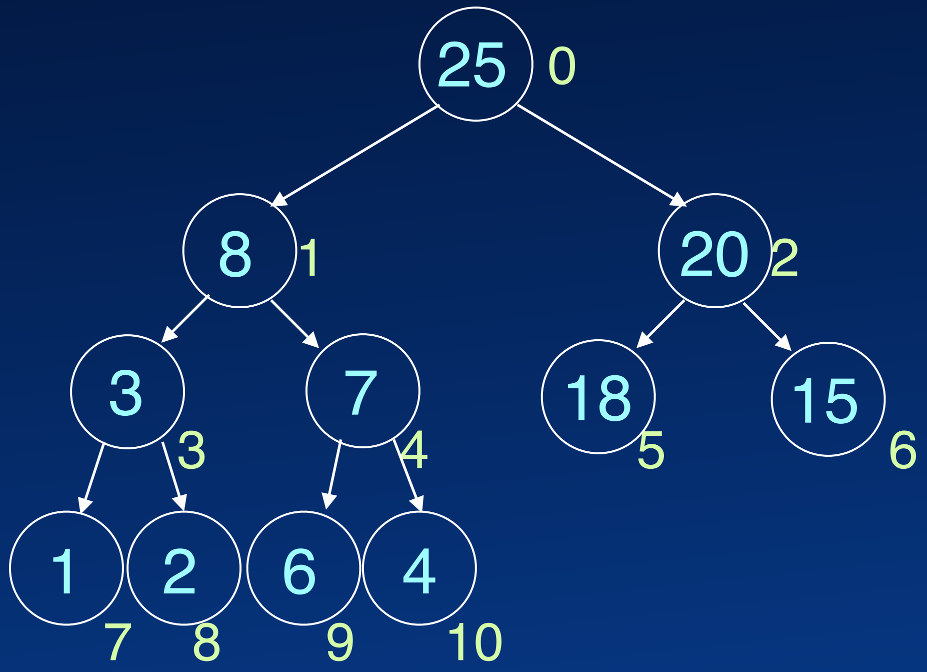


Heap



Heap

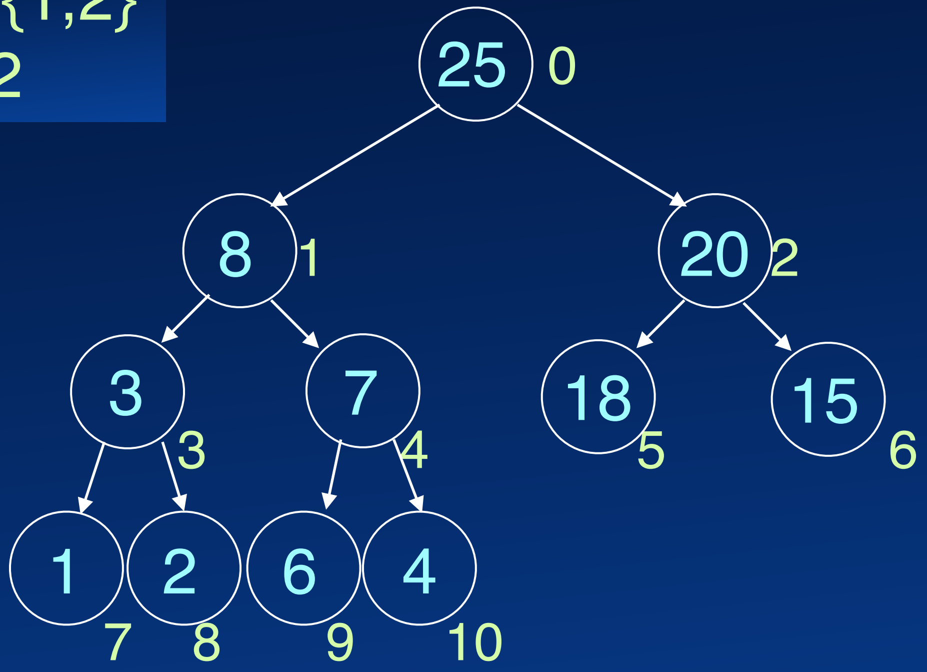
Heap



Heap

Heap

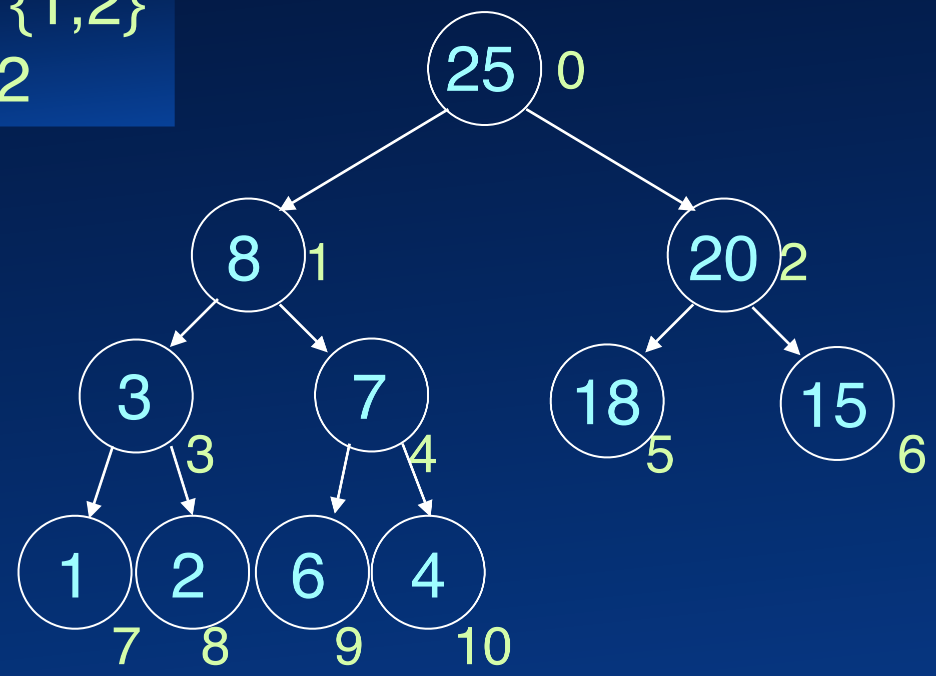
child-index = 2*parent-index + {1,2}
parent-index = (child-index-1)/2



Heap

Heap

child-index = 2*parent-index + {1,2}
parent-index = (child-index-1)/2

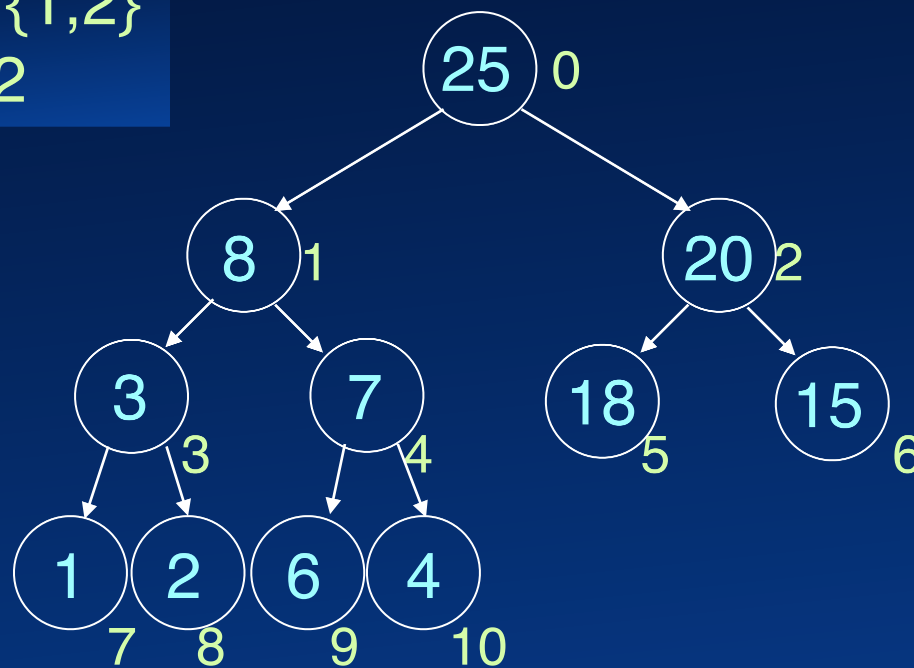


25	8	20	3	7	18	15	1	2	6	4		

Heap

Heap

child-index = 2*parent-index + {1,2}
parent-index = (child-index-1)/2

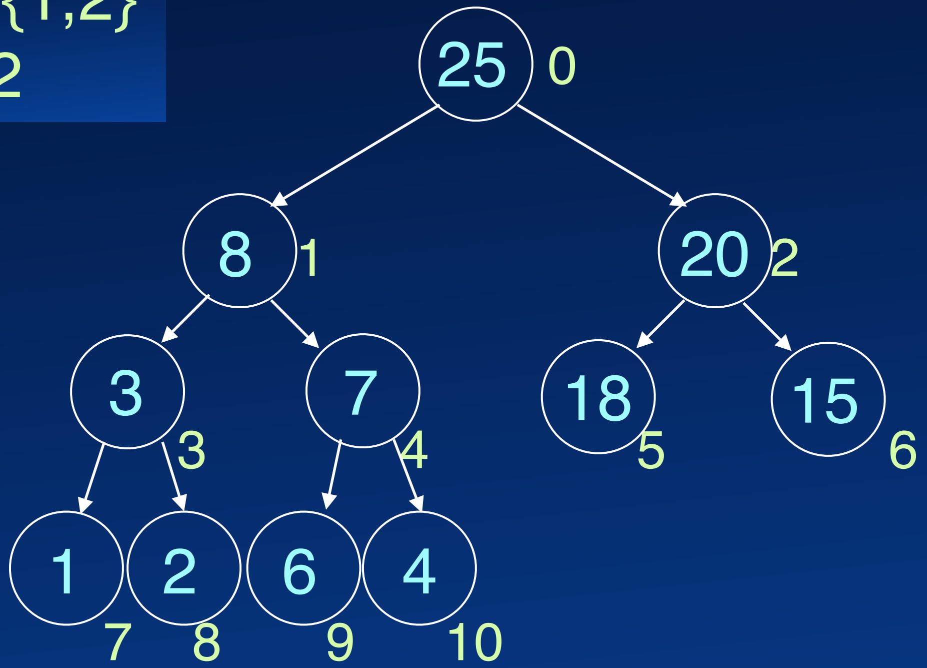


25	8	20	3	7	18	15	1	2	6	4	
0	1	2	3	4	5	6	7	8	9	10	11

Heap

Heap

child-index = 2*parent-index + {1,2}
parent-index = (child-index-1)/2

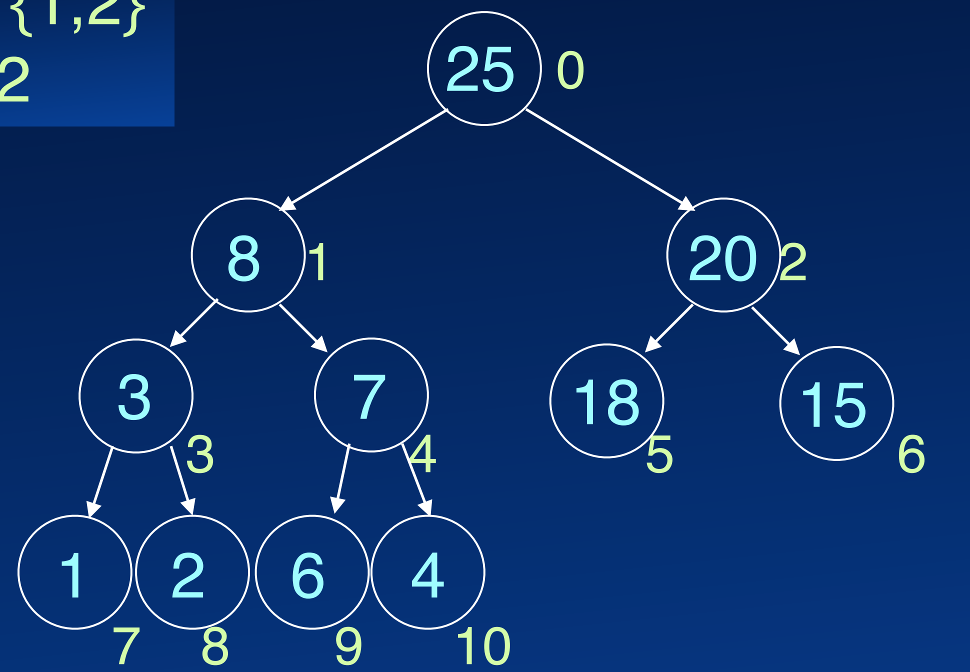


25	8	20	3	7	18	15	1	2	6	4		
0	1	2	3	4	5	6	7	8	9	10	11	

Heap

Heap

child-index = 2*parent-index + {1,2}
parent-index = (child-index-1)/2



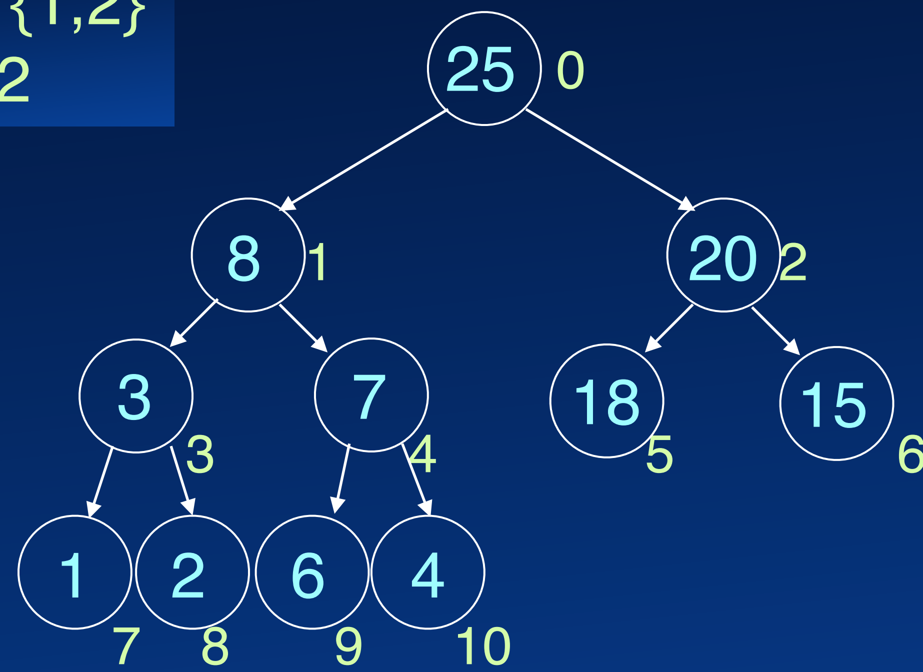
Heap

25	8	20	3	7	18	15	1	2	6	4		
0	1	2	3	4	5	6	7	8	9	10	11	

↑ next

Heap

child-index = 2*parent-index + {1,2}
parent-index = (child-index-1)/2



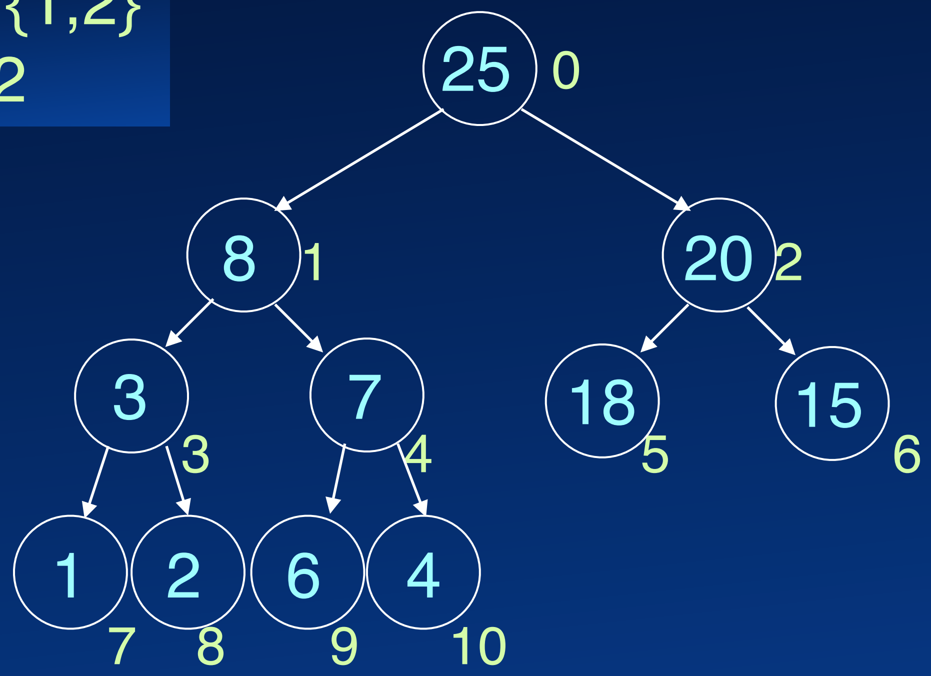
Heap

25	8	20	3	7	18	15	1	2	6	4		
0	1	2	3	4	5	6	7	8	9	10	11	

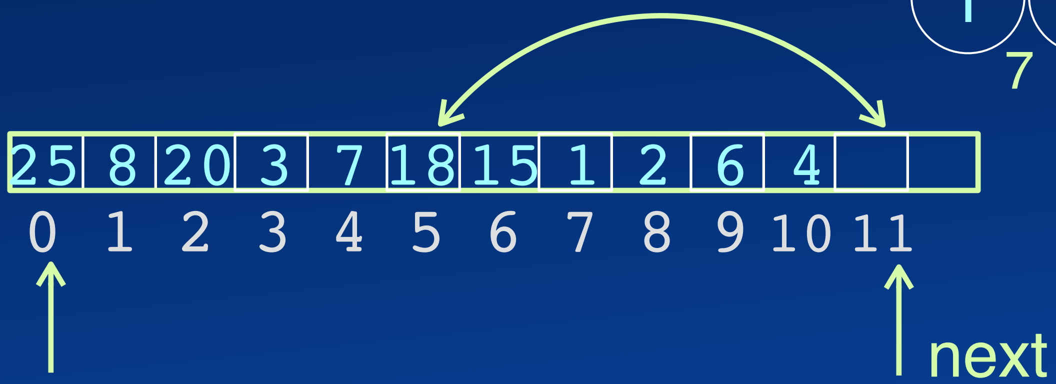
next

Heap

child-index = 2*parent-index + {1,2}
parent-index = (child-index-1)/2

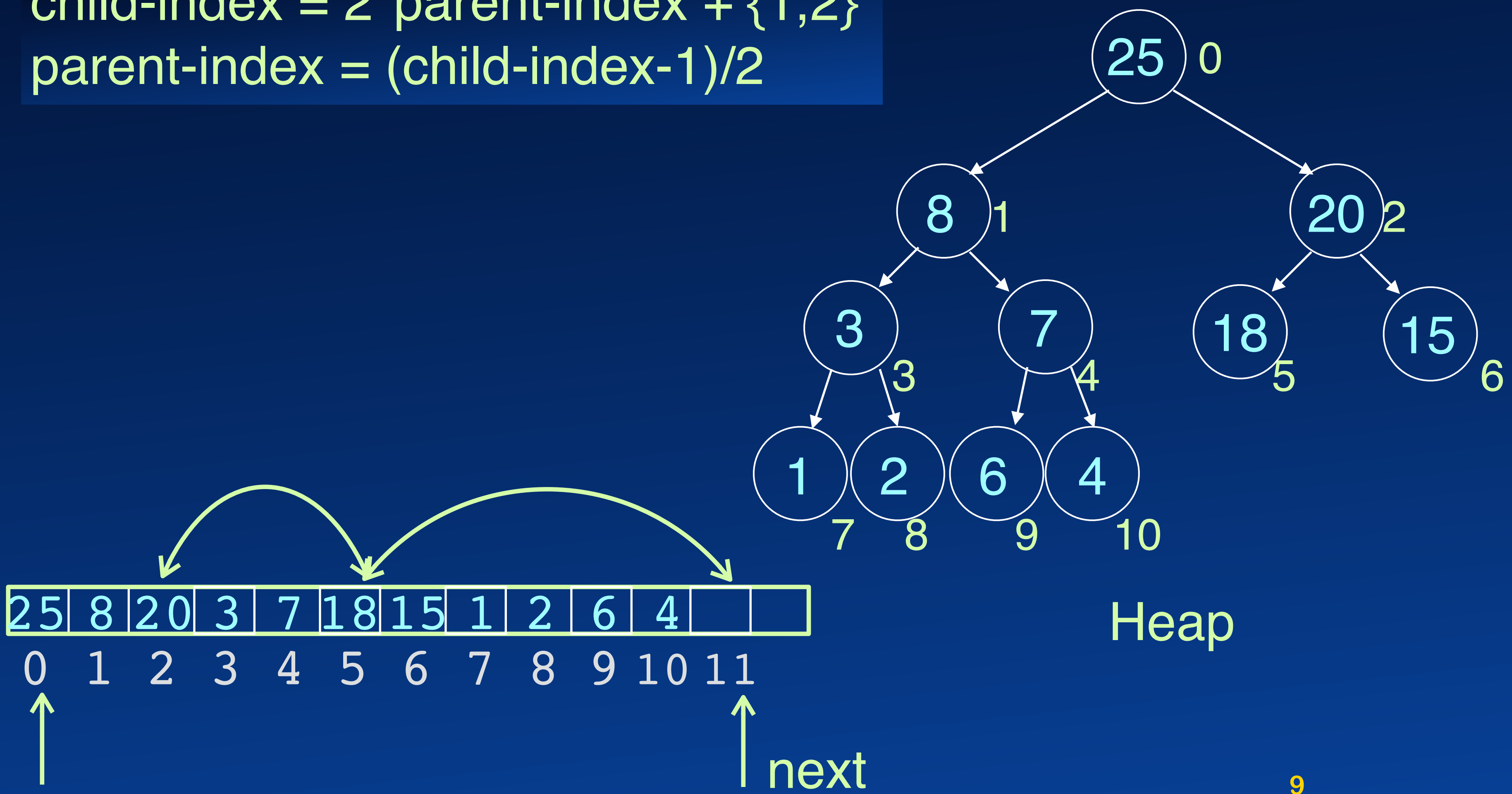


Heap



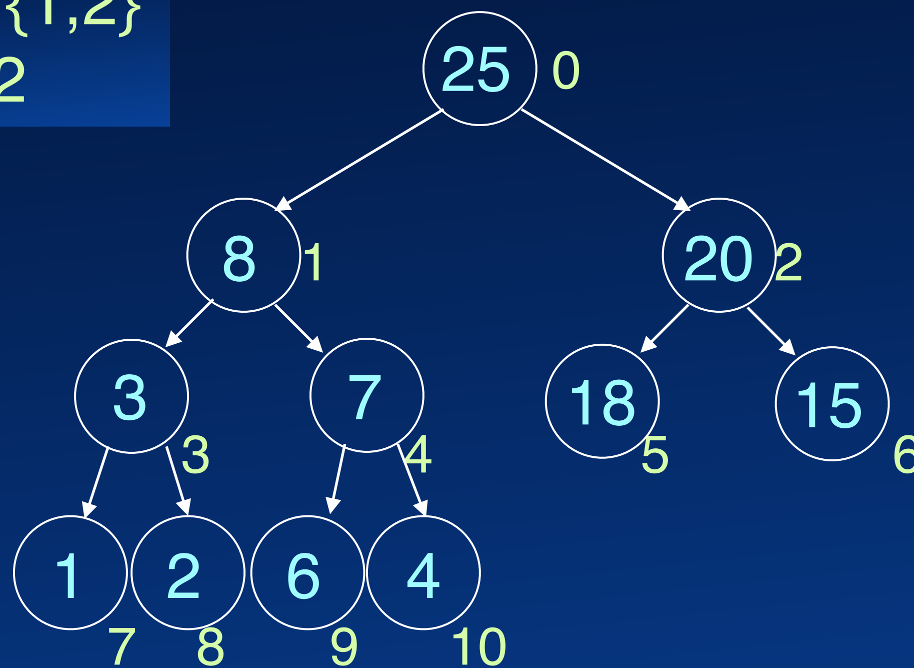
Heap

child-index = 2*parent-index + {1,2}
parent-index = (child-index-1)/2

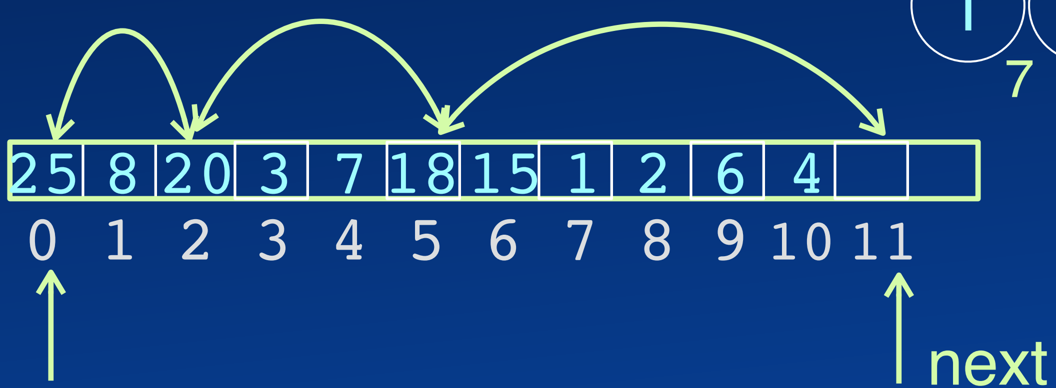


Heap

child-index = 2*parent-index + {1,2}
parent-index = (child-index-1)/2

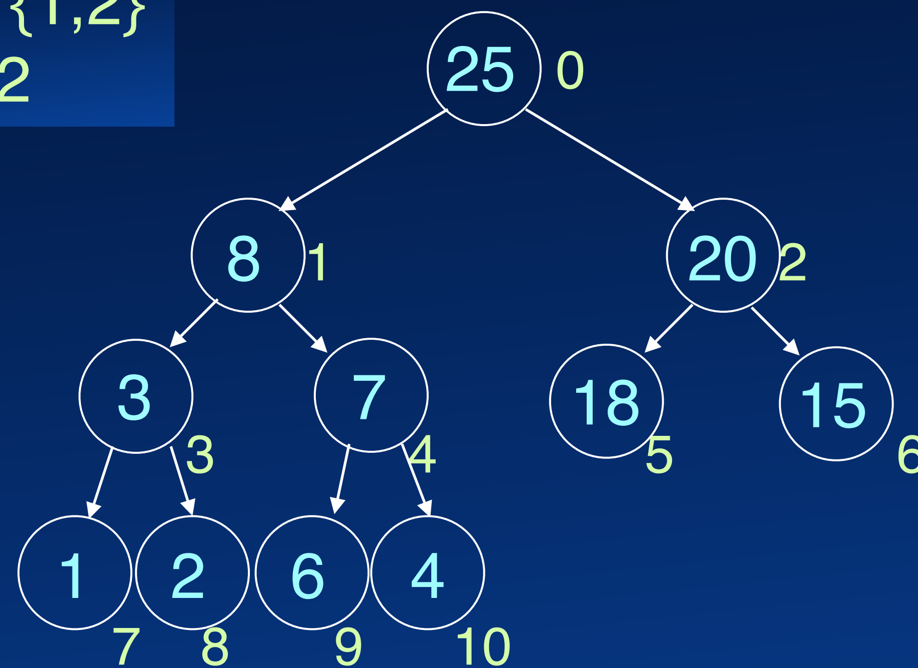


Heap



Heap

child-index = 2*parent-index + {1,2}
parent-index = (child-index-1)/2



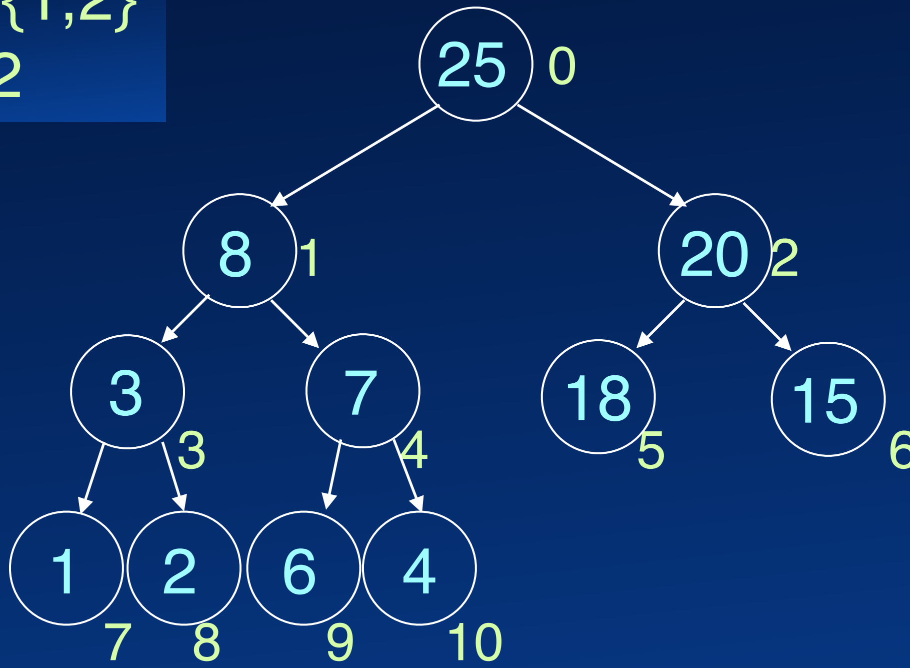
Heap

25	8	20	3	7	18	15	1	2	6	4	
0	1	2	3	4	5	6	7	8	9	10	11

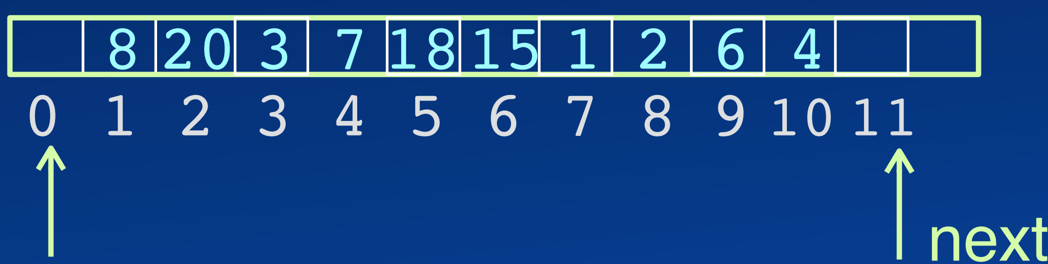
next

Heap

child-index = 2*parent-index + {1,2}
parent-index = (child-index-1)/2

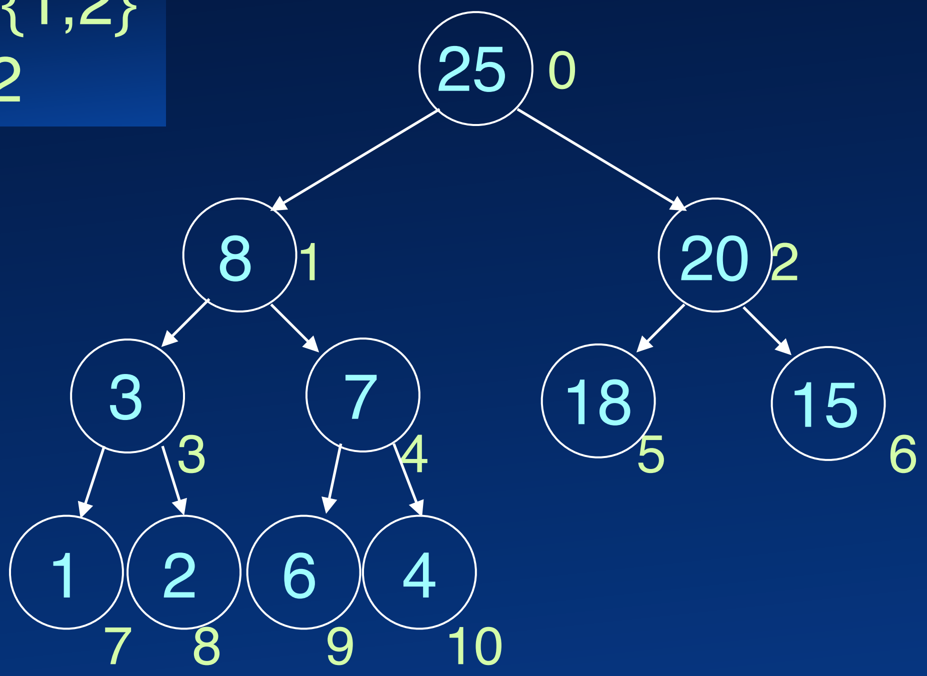


Heap

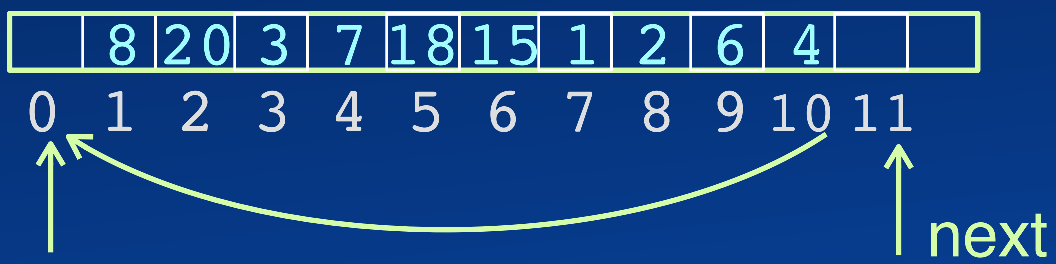


Heap

child-index = 2*parent-index + {1,2}
parent-index = (child-index-1)/2

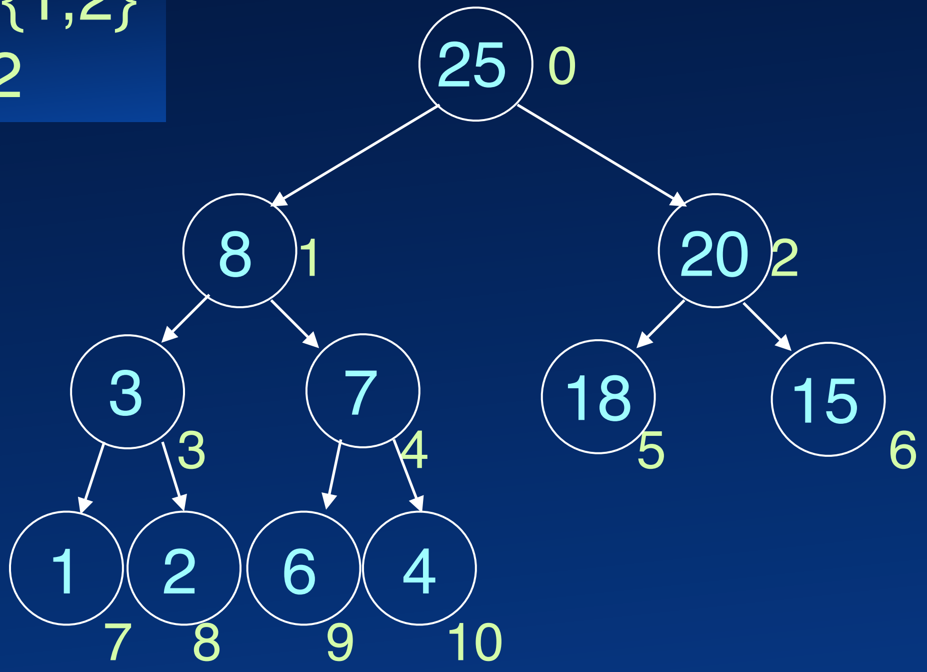


Heap



Heap

child-index = 2*parent-index + {1,2}
parent-index = (child-index-1)/2



Heap