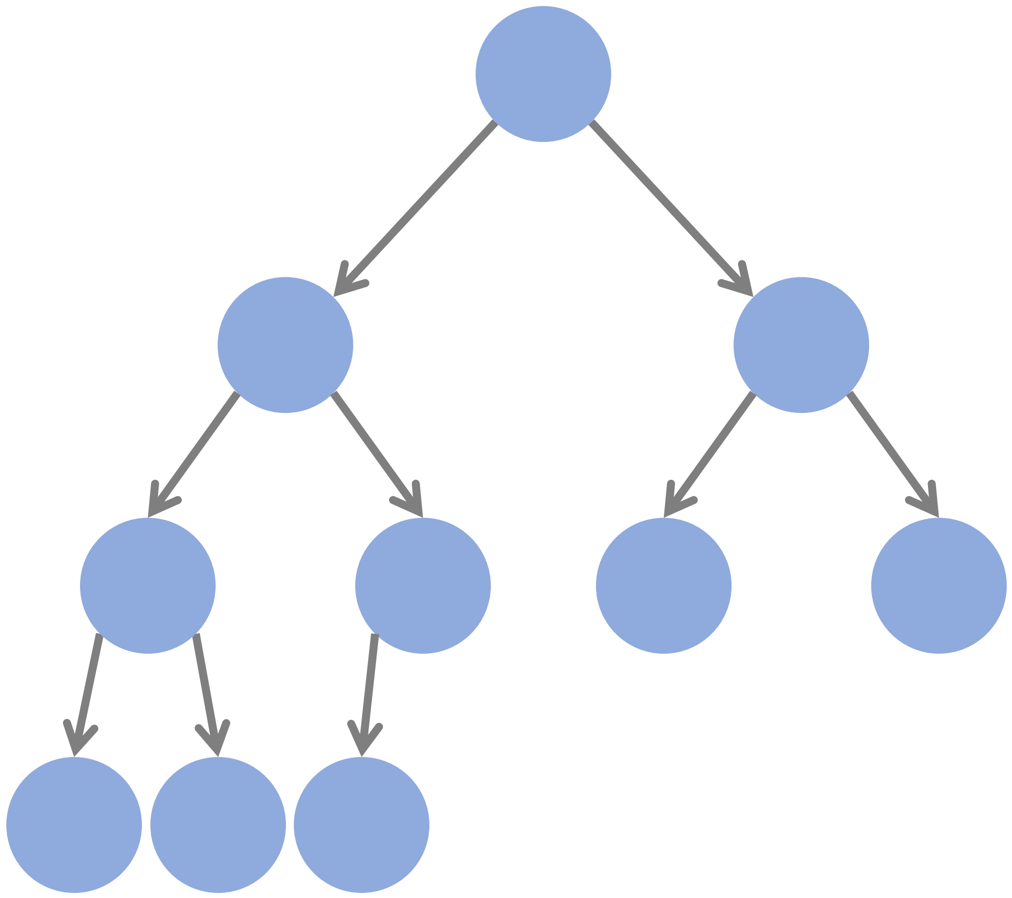
Binary Heaps

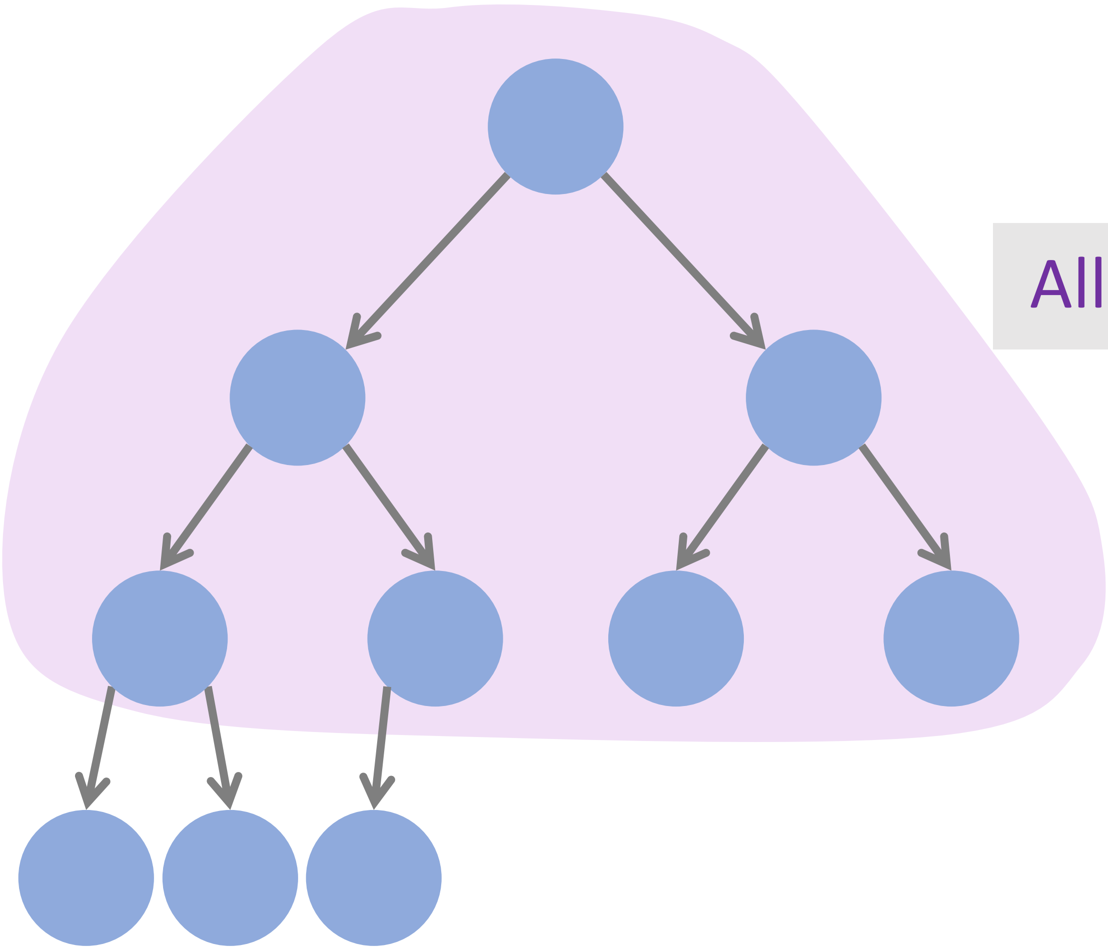
Shusen Wang

Complete Binary Trees

Complete Binary Tree

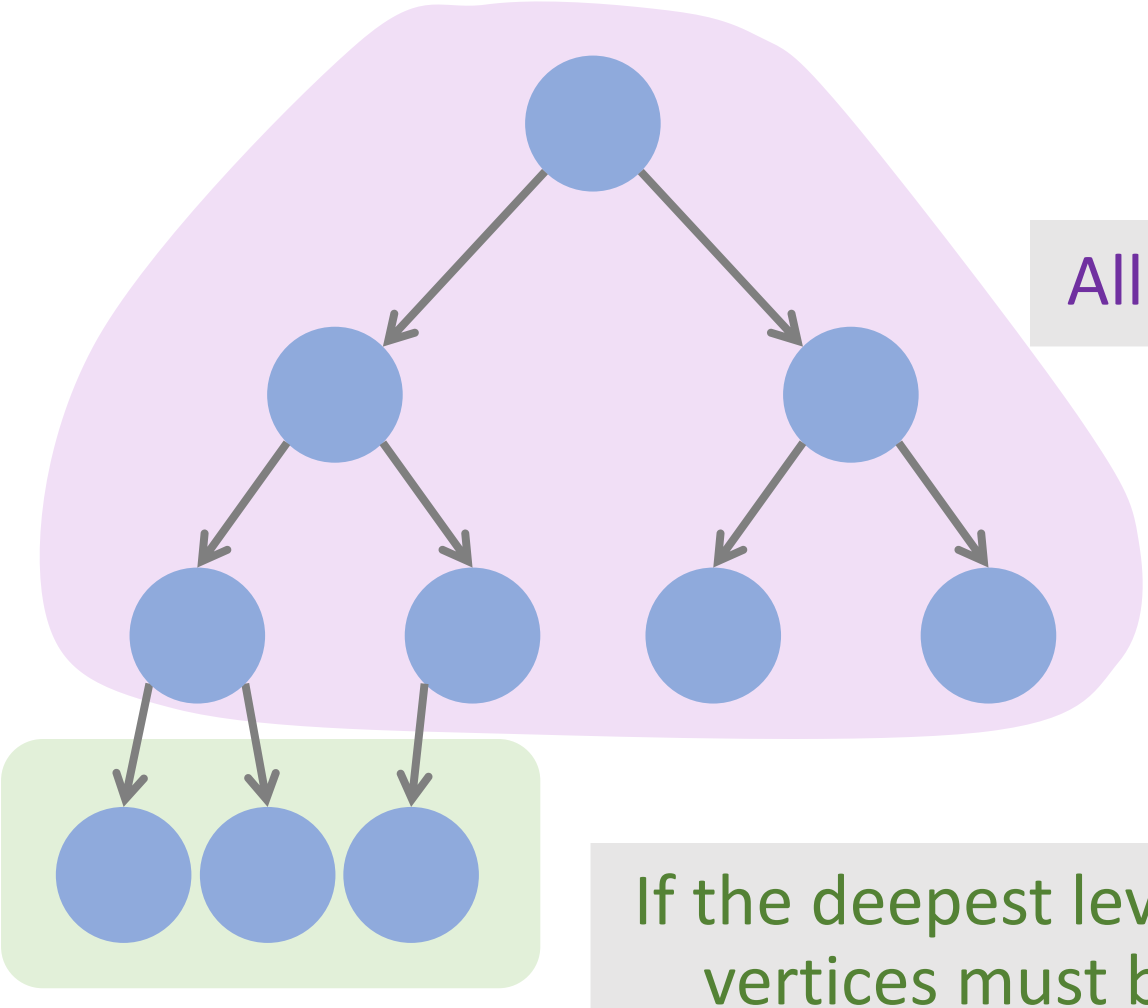


Complete Binary Tree



All levels, except the deepest, are fully filled.

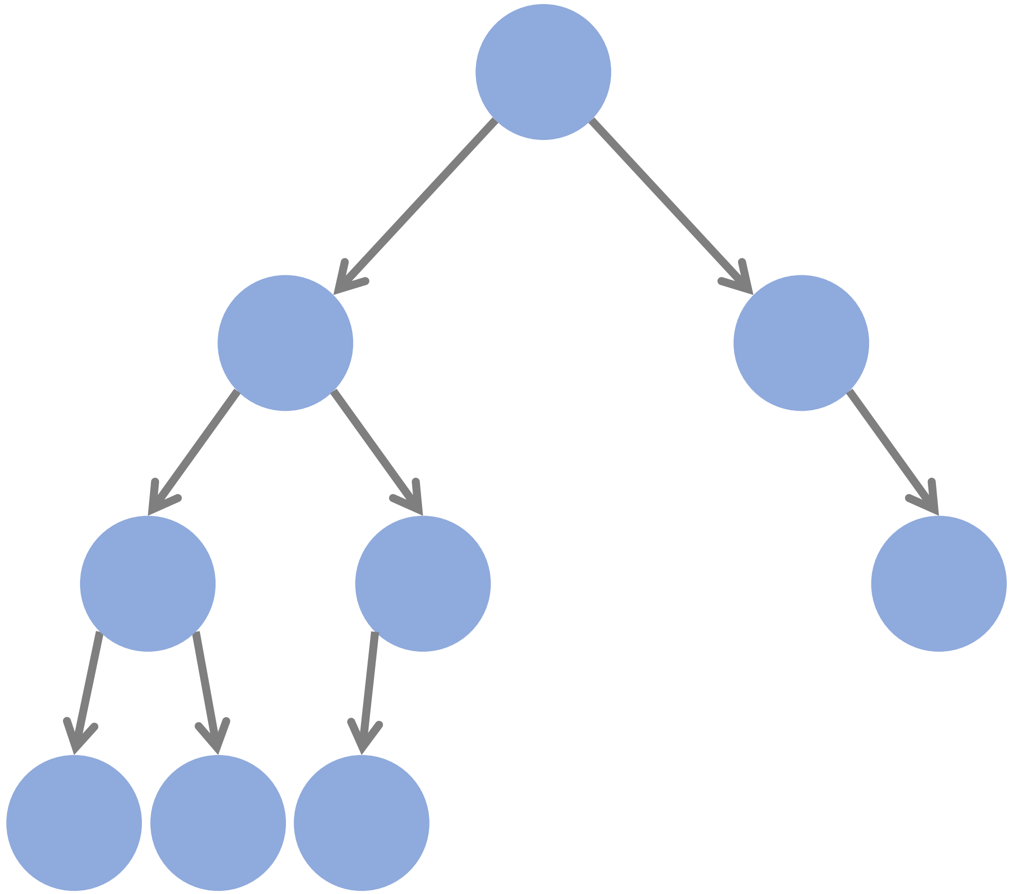
Complete Binary Tree



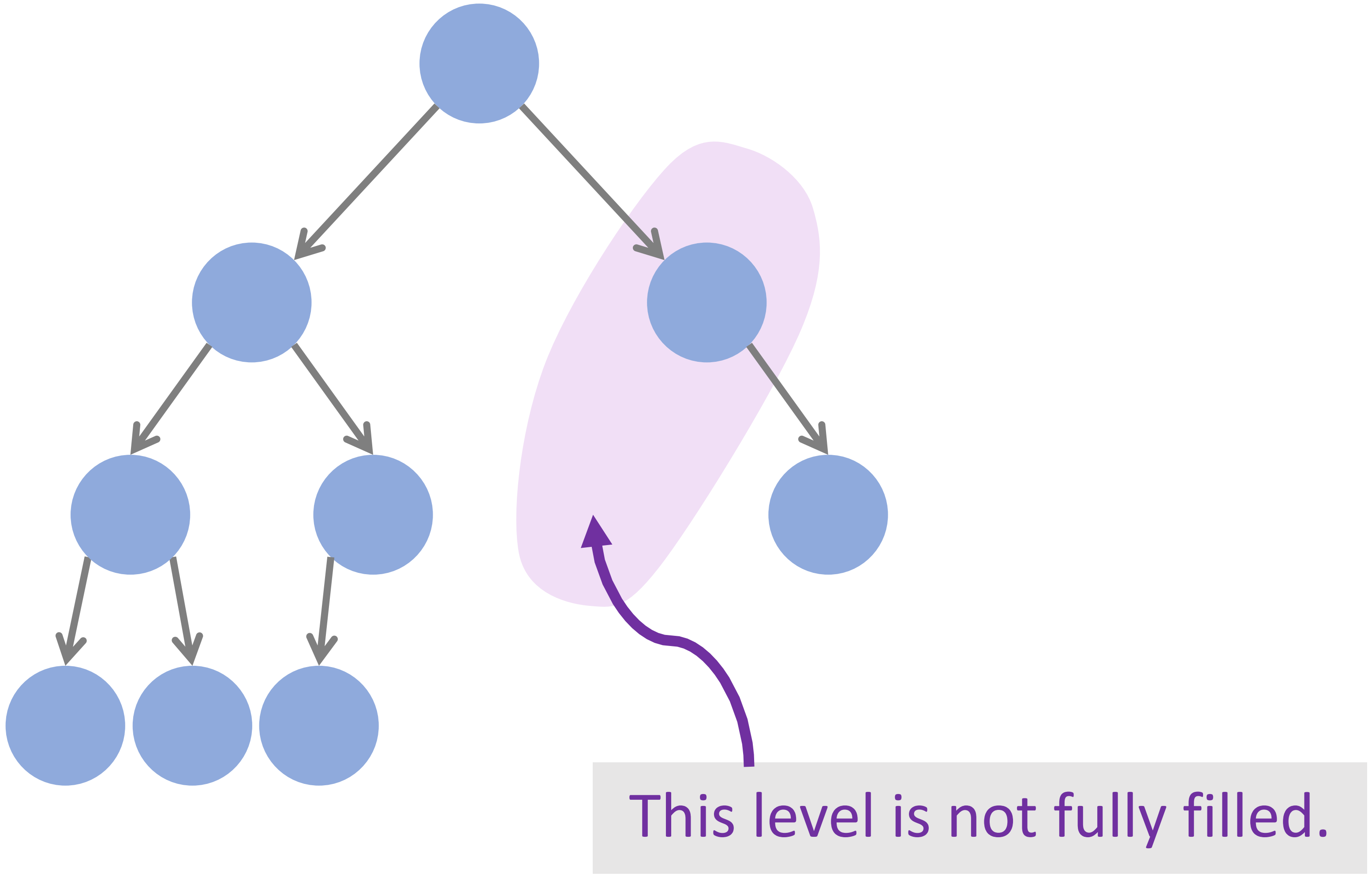
All levels, except the deepest, are fully filled.

If the deepest level is not complete, then the vertices must be filled **from left to right**.

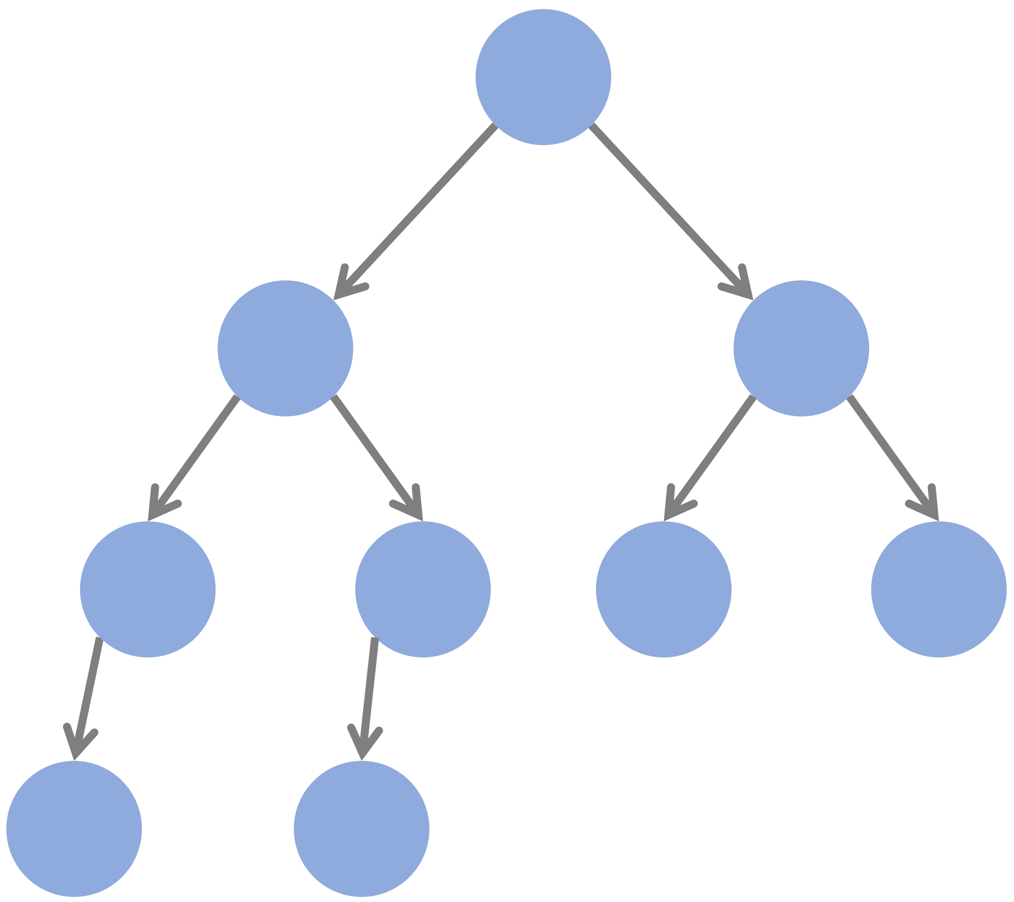
Is this a complete binary tree



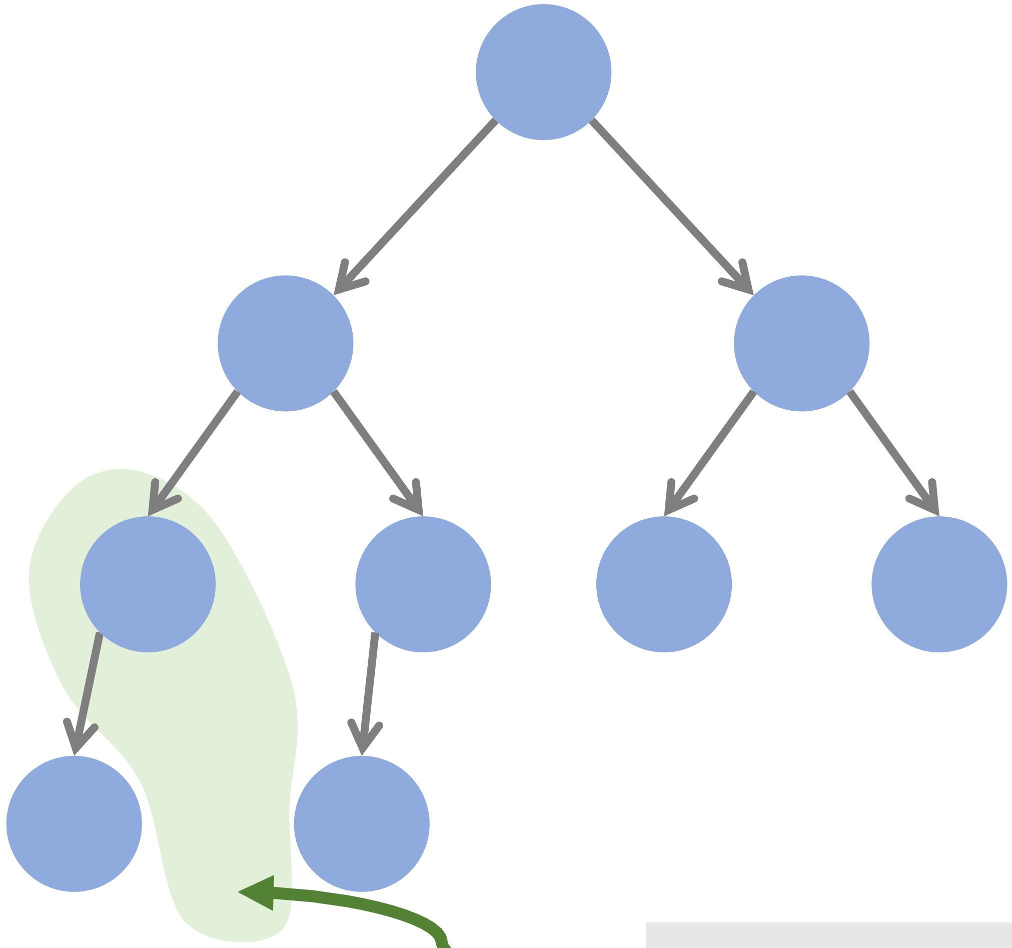
This is not complete binary tree



Is this a complete binary tree

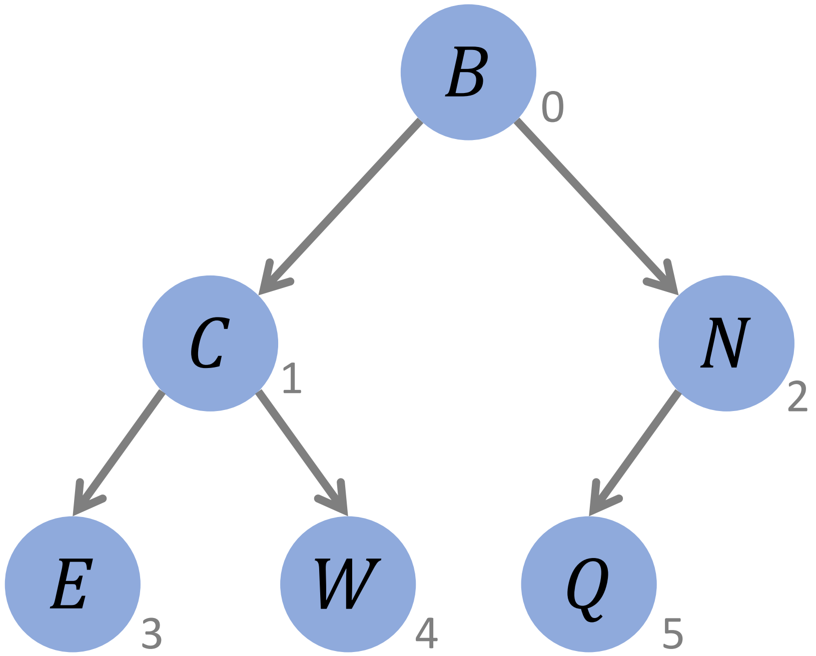


This is not complete binary tree



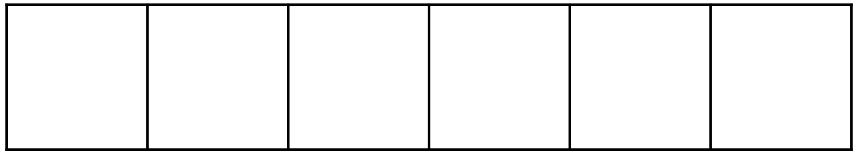
The deepest level is not filled from left to right.

Store complete binary tree in array

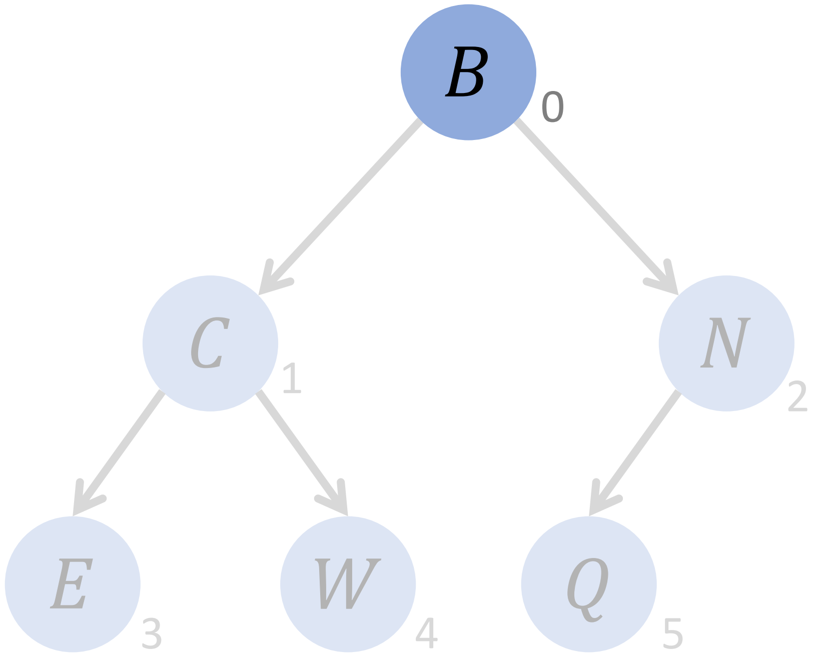


Indices:

Keys:

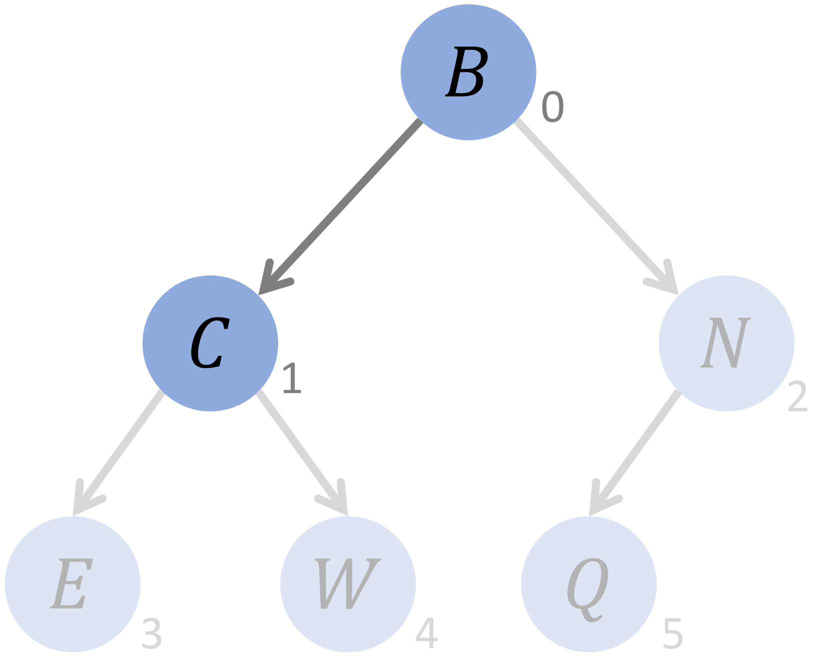


Store complete binary tree in array



Indices:	0					
Keys:	B					

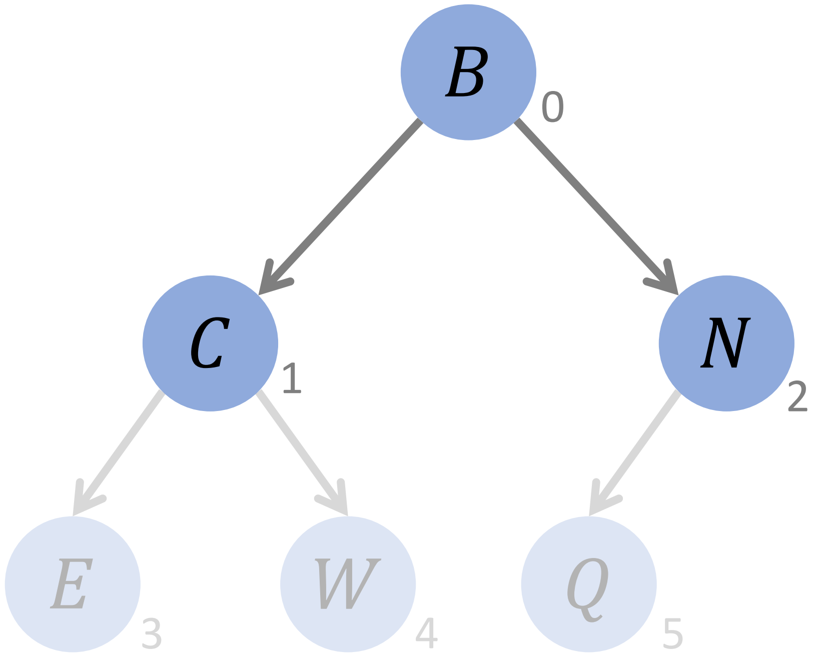
Store complete binary tree in array



Indices: 0 1

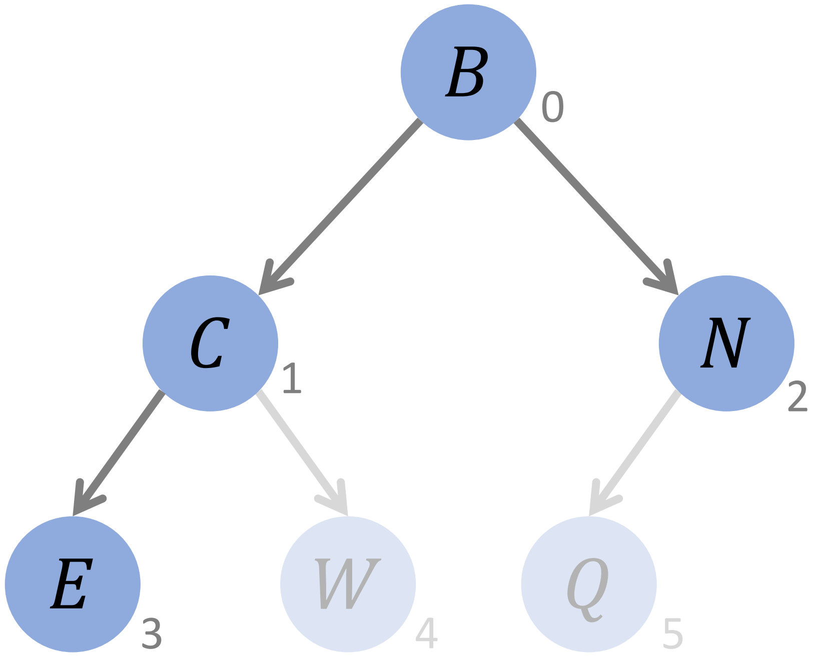
Keys:	B	C				

Store complete binary tree in array



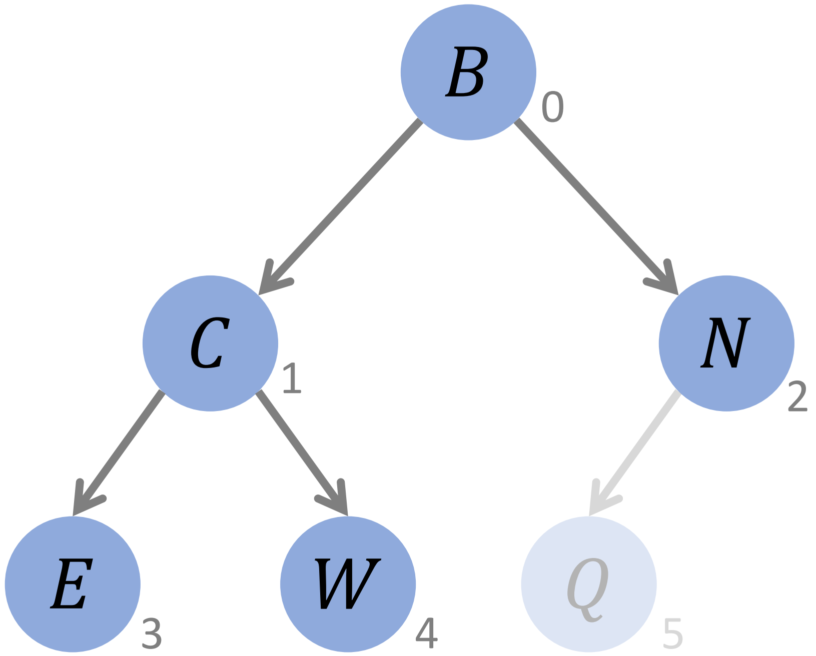
Indices:	0	1	2			
Keys:	B	C	N			

Store complete binary tree in array



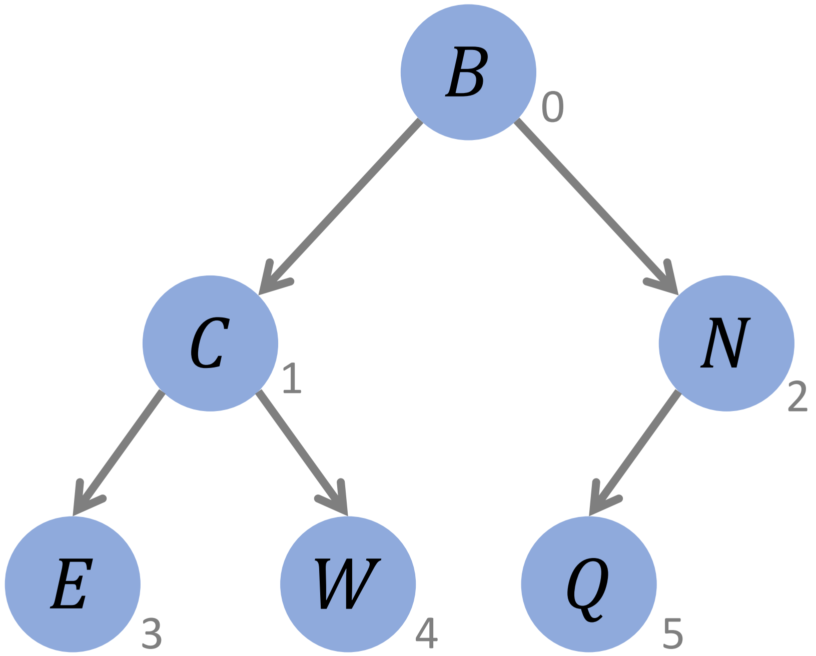
Indices:	0	1	2	3		
Keys:	B	C	N	E		

Store complete binary tree in array



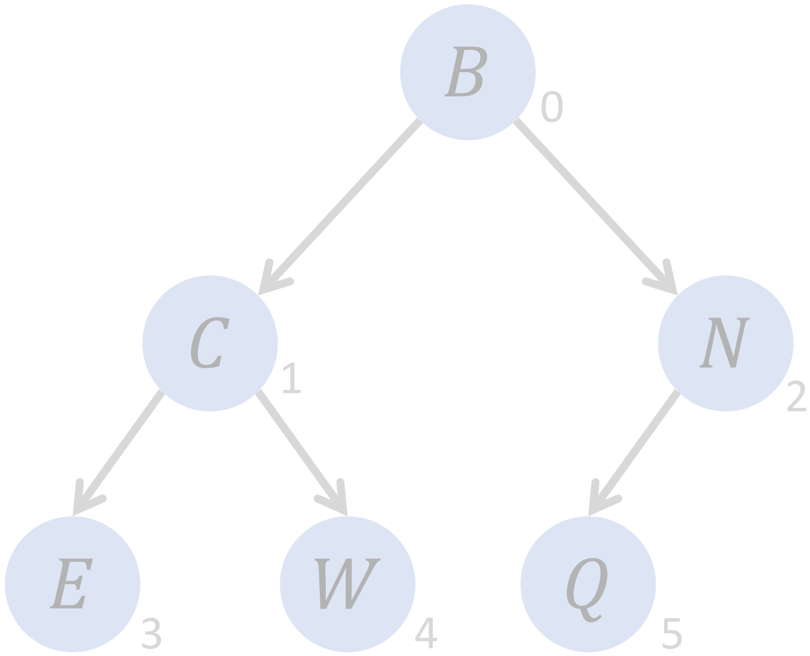
Indices:	0	1	2	3	4	
Keys:	B	C	N	E	W	

Store complete binary tree in array



Indices:	0	1	2	3	4	5
Keys:	B	C	N	E	W	Q

Store complete binary tree in array



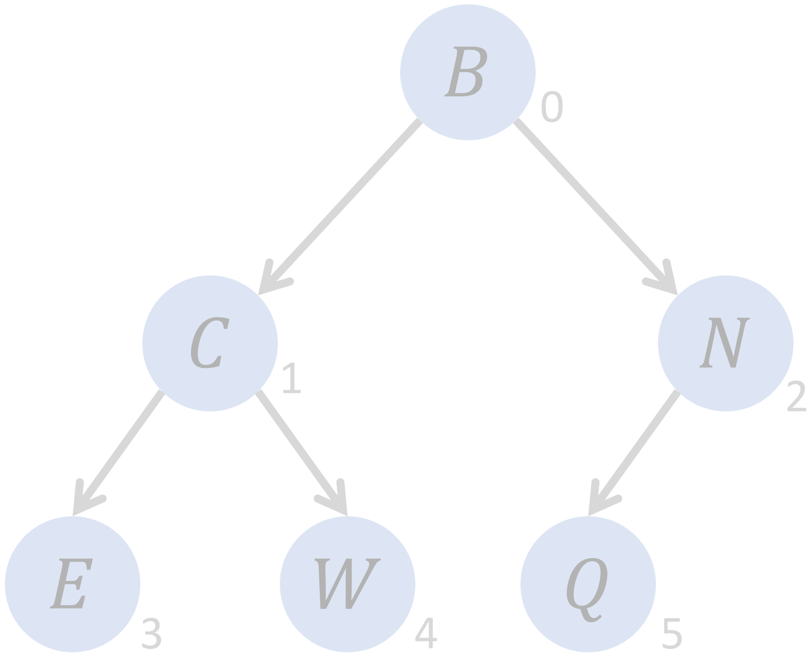
Indices:	0	1	2	3	4	5
Keys:	B	C	N	E	W	Q

Find children

- A vertex's index is i.
- Its children's indices are

  $2i + 1$ and $2i + 2$.

Store complete binary tree in array



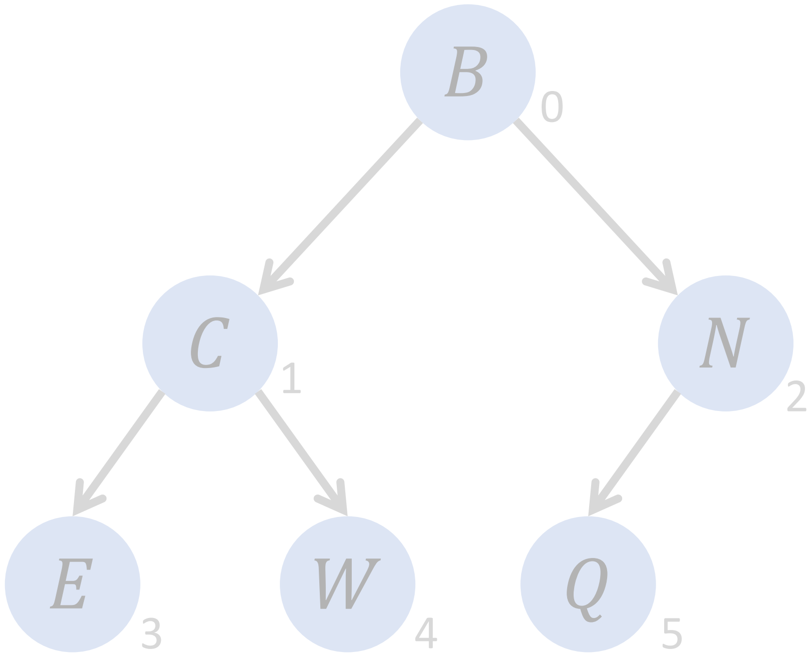
Find children

- A vertex's index is i.
- Its children's indices are

  $2i+1$ and $2i+2$.

Indices:	0	1	2	3	4	5
Keys:	B	C	N	E	W	Q

Store complete binary tree in array



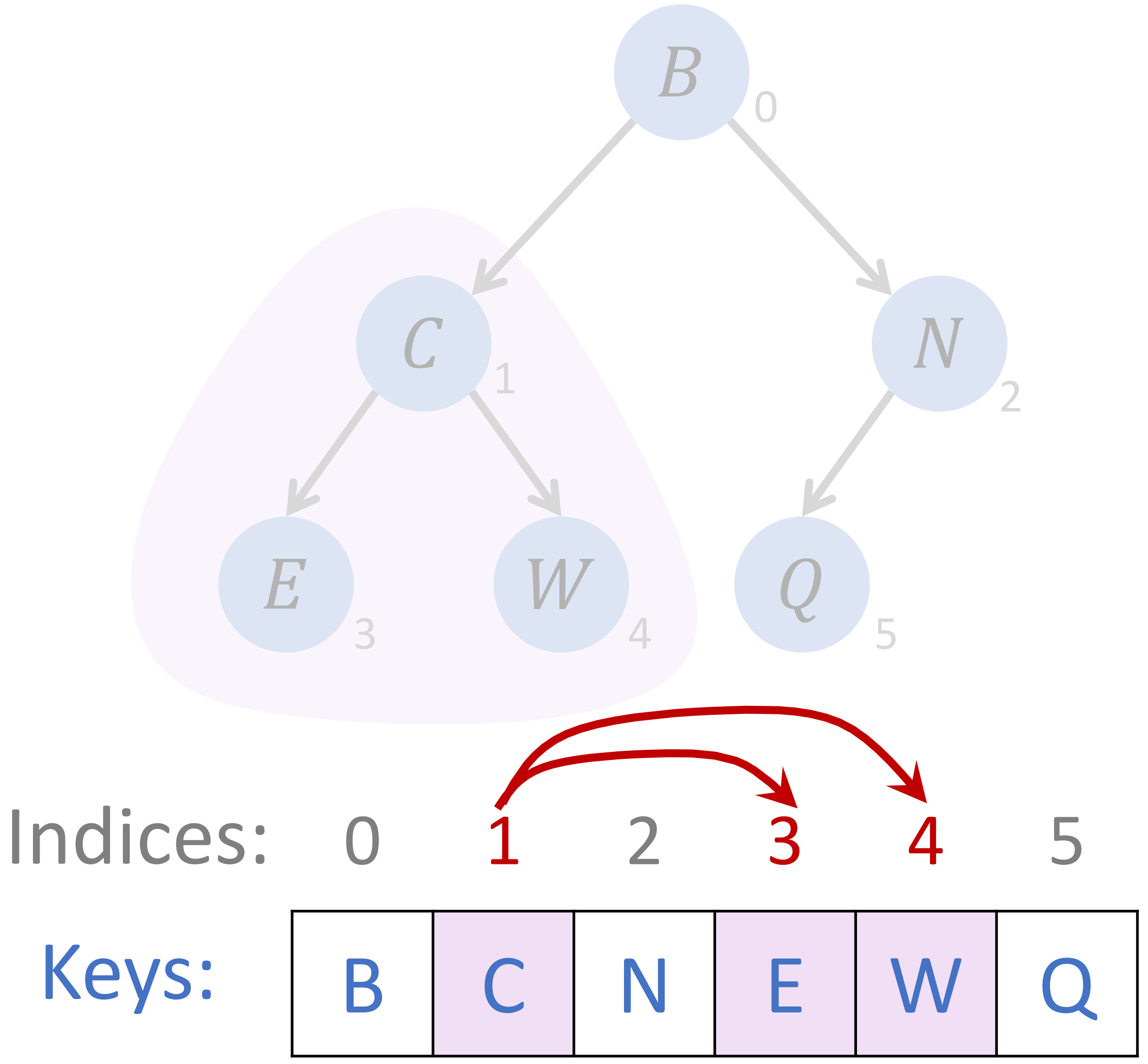
Indices:	0	1	2	3	4	5
Keys:	B	C	N	E	W	Q

Find children

- A vertex's index is i.
- Its children's indices are

  $2i + 1$ and $2i + 2$.

Store complete binary tree in array

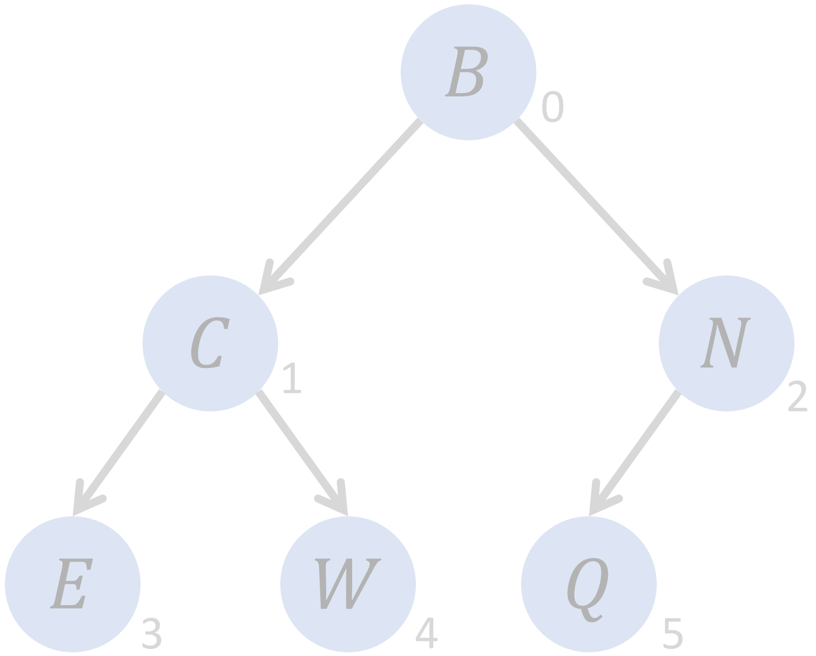


Find children

- A vertex's index is i.
- Its children's indices are

  $2i + 1$ and $2i + 2$.

Store complete binary tree in array



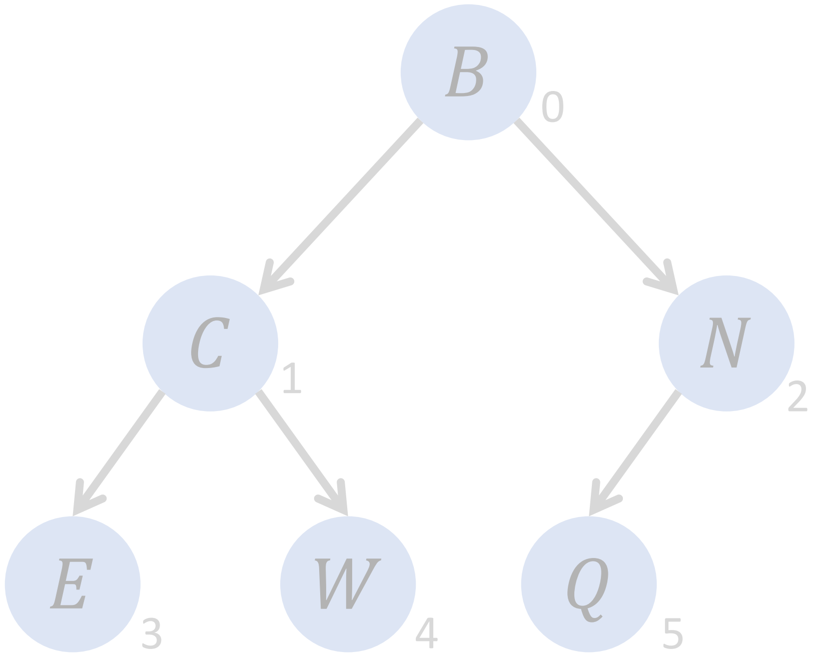
Indices: 0 1 2 3 4 5

Keys:					
B	C	N	E	W	Q

Find parent

- A vertex's index is j.
- Its parent's index is $\left\lfloor \frac{j-1}{2} \right\rfloor$.

Store complete binary tree in array

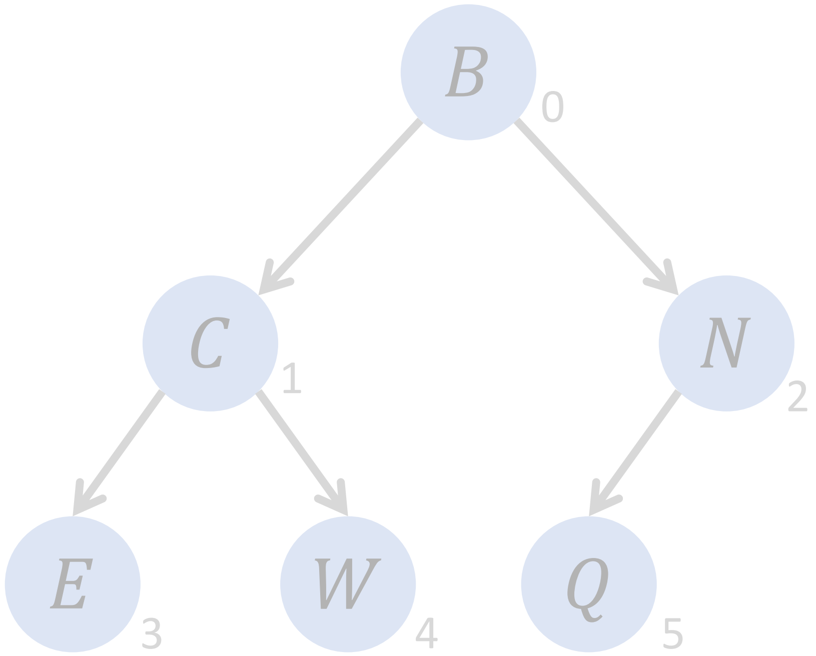


Indices:	0	1	2	3	4	5
Keys:	B	C	N	E	W	Q

Find parent

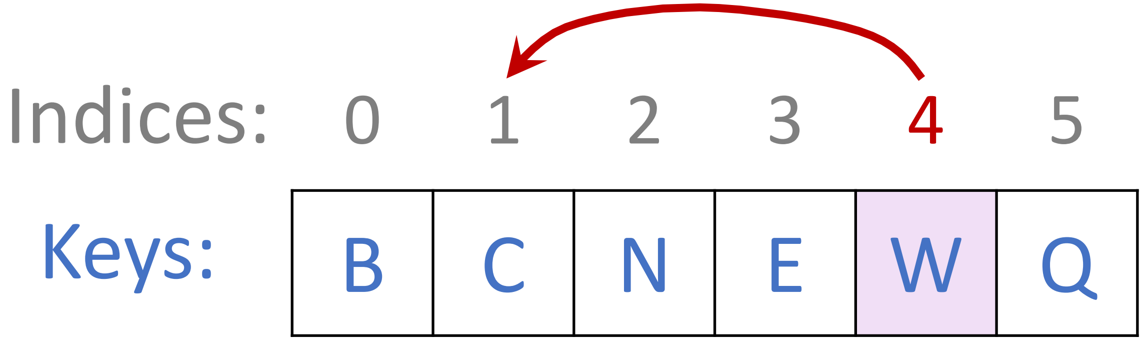
- A vertex's index is j.
- Its parent's index is $\left\lfloor \frac{j-1}{2} \right\rfloor$.

Store complete binary tree in array

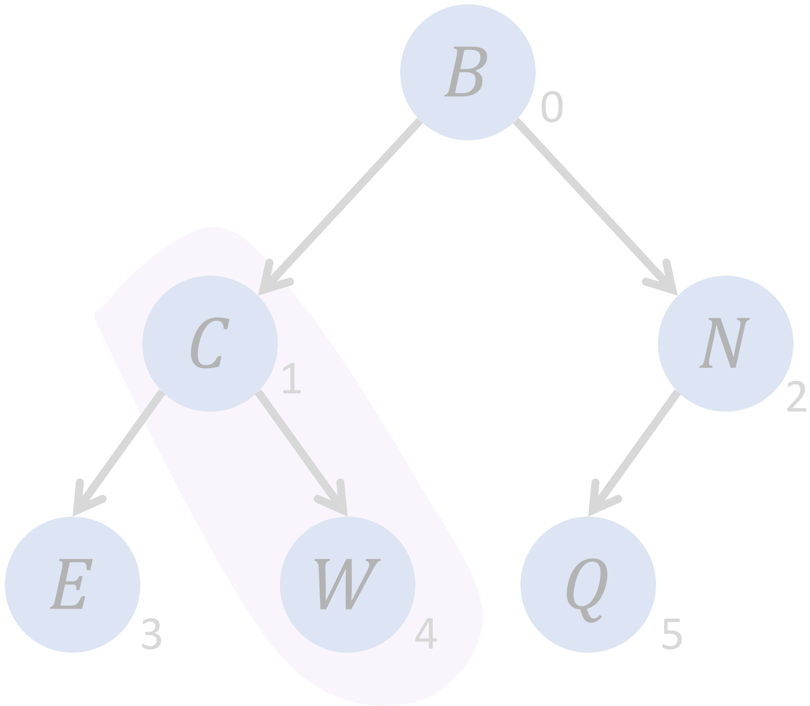


Find parent

- A vertex's index is j.
- Its parent's index is $\left\lfloor \frac{j-1}{2} \right\rfloor$.



Store complete binary tree in array



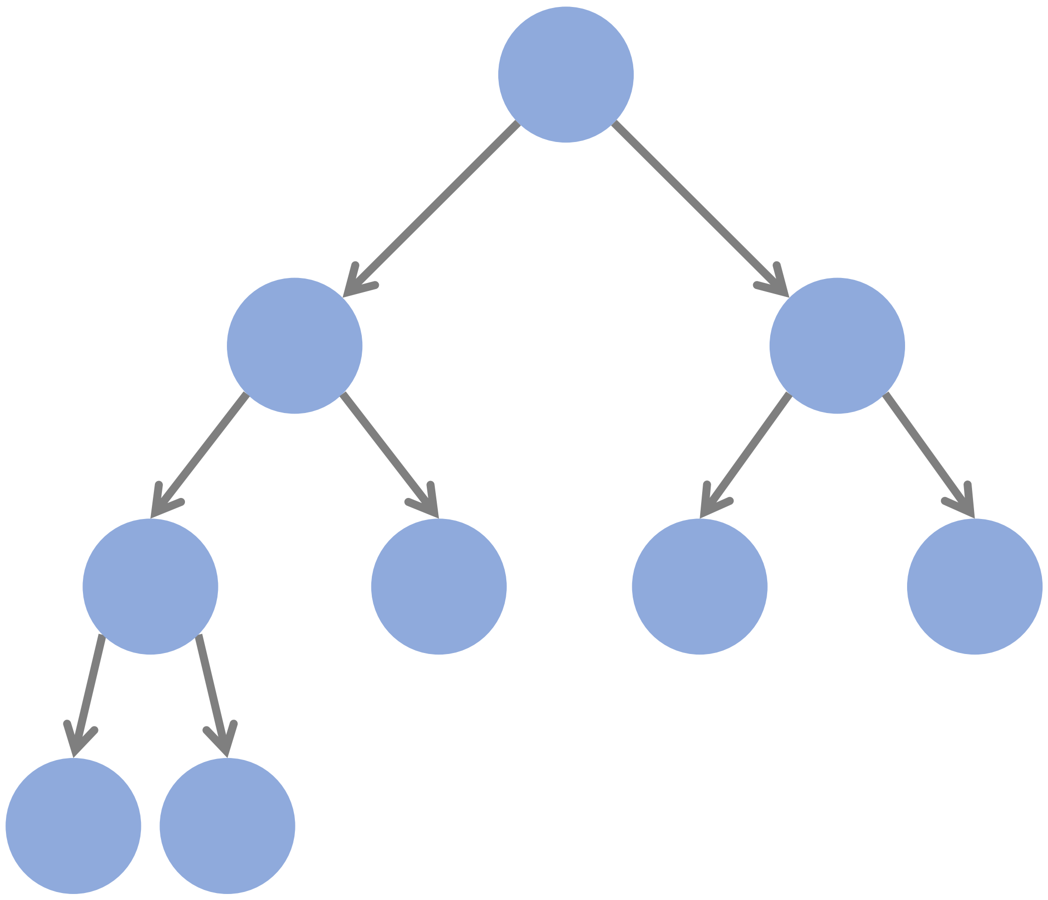
Indices:	0	1	2	3	4	5
Keys:	B	C	N	E	W	Q

Find parent

- A vertex's index is j.
- Its parent's index is $\left\lfloor \frac{j-1}{2} \right\rfloor$.

Binary Heaps

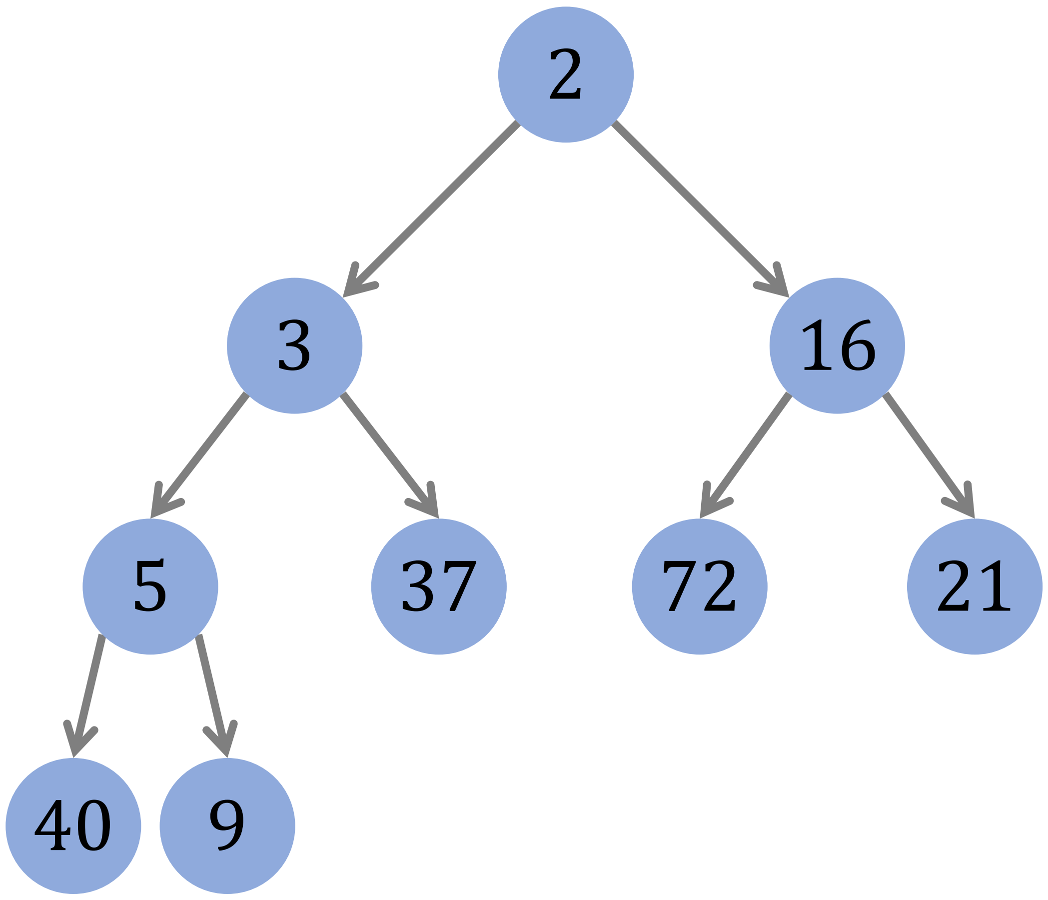
Binary Heaps



Properties

- Binary heaps are complete binary trees.

Binary Heaps

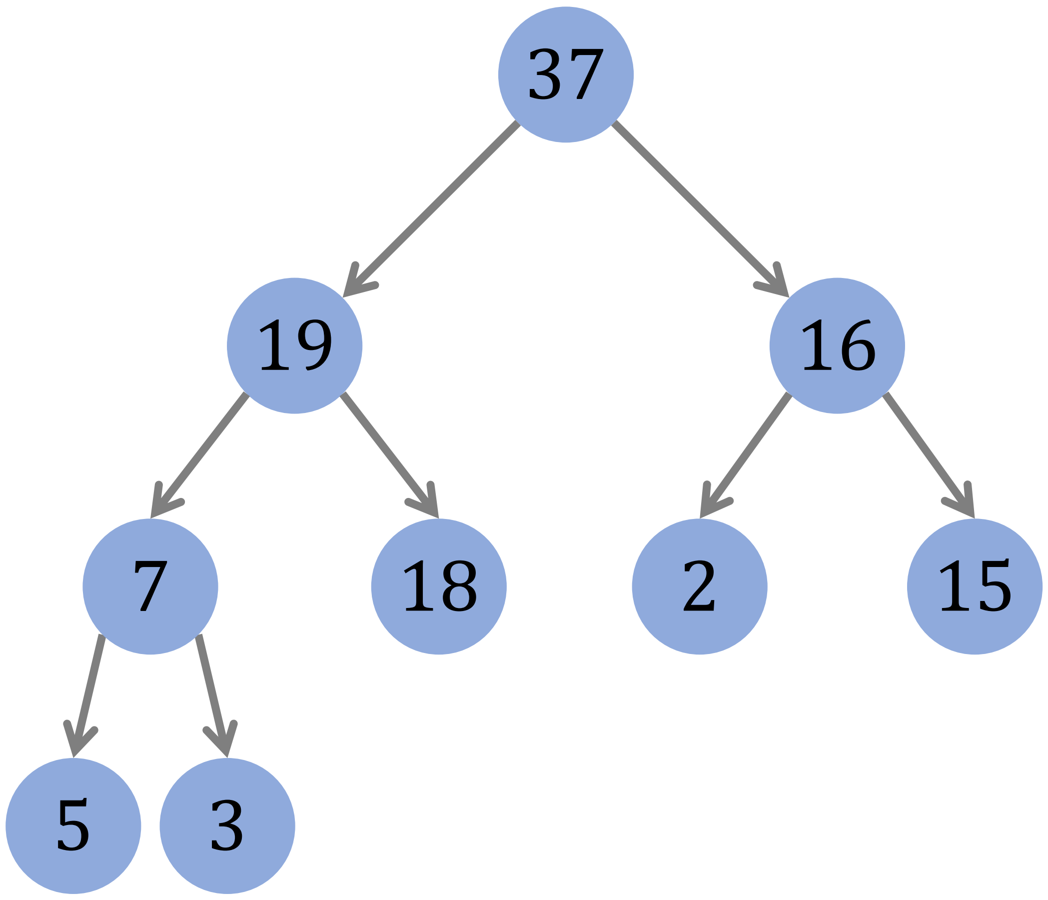


Min-heap

Properties

- Binary heaps are complete binary trees.
- **Min-heap**: parent's key $\leq$ children's keys.

Binary Heaps

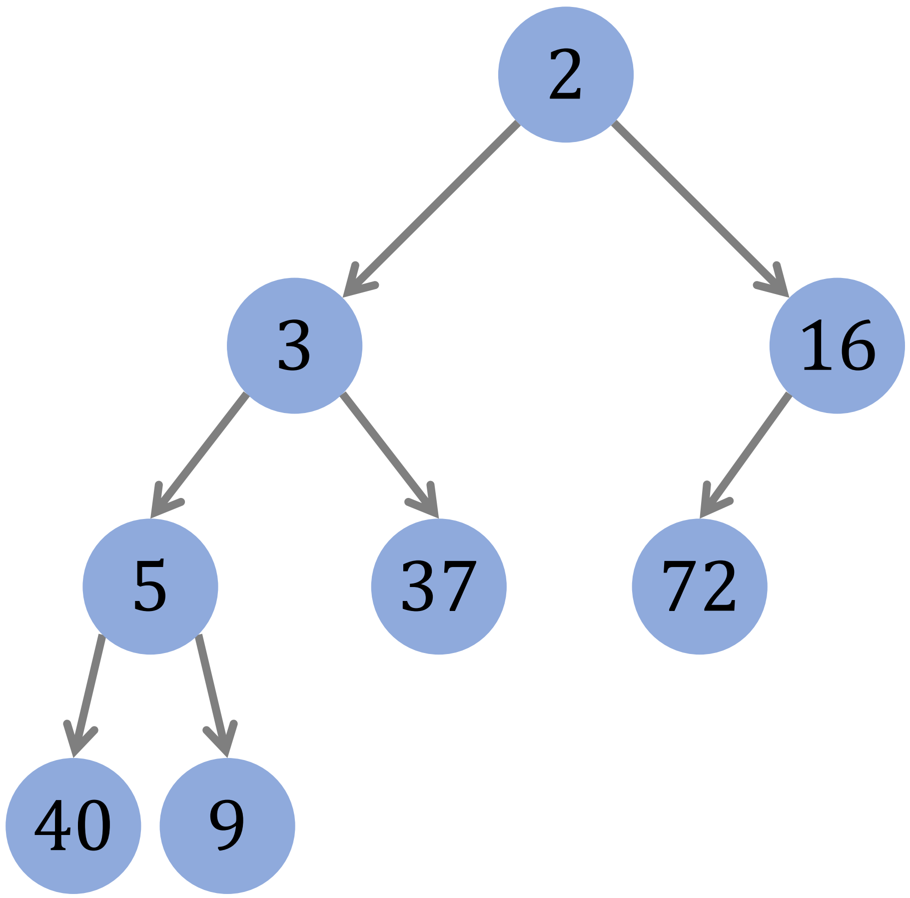


Max-heap

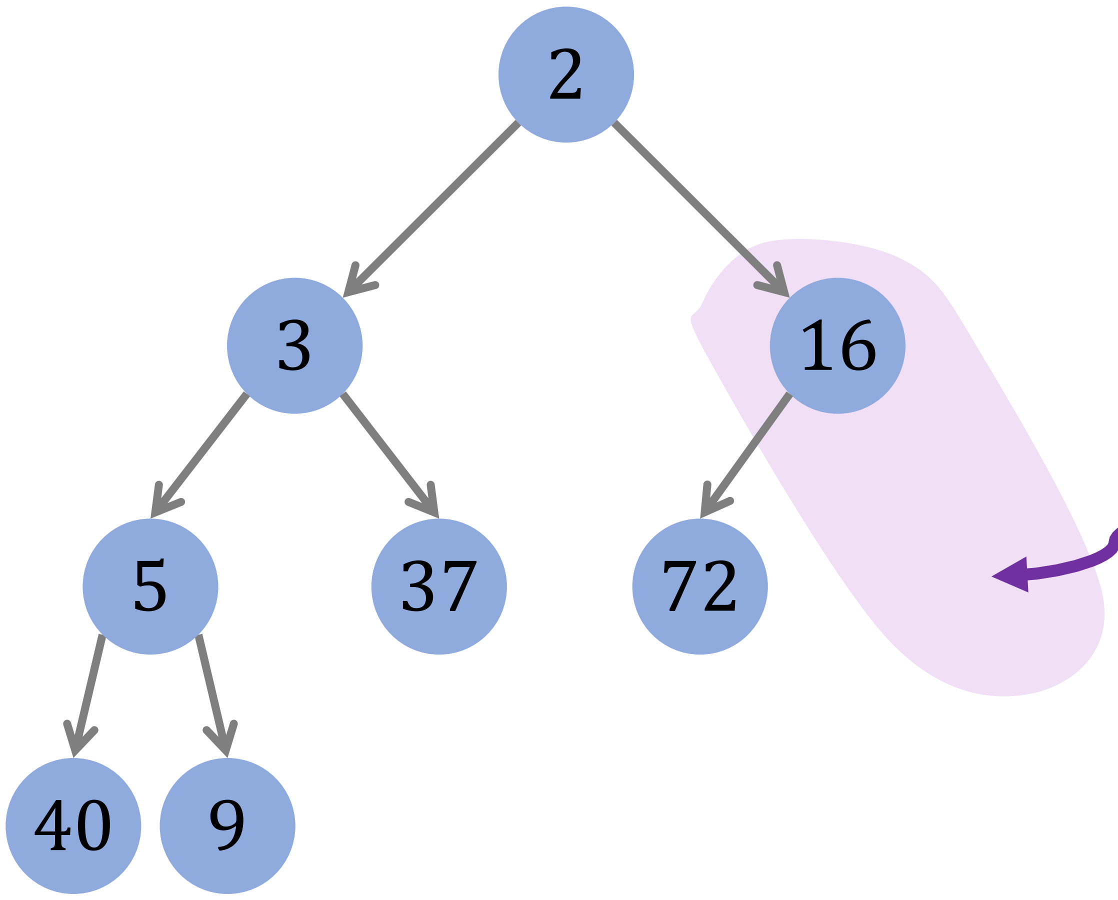
Properties

- Binary heaps are complete binary trees.
- **Max-heap**: parent's key $\geq$ children's keys.

Is this a min-heap?

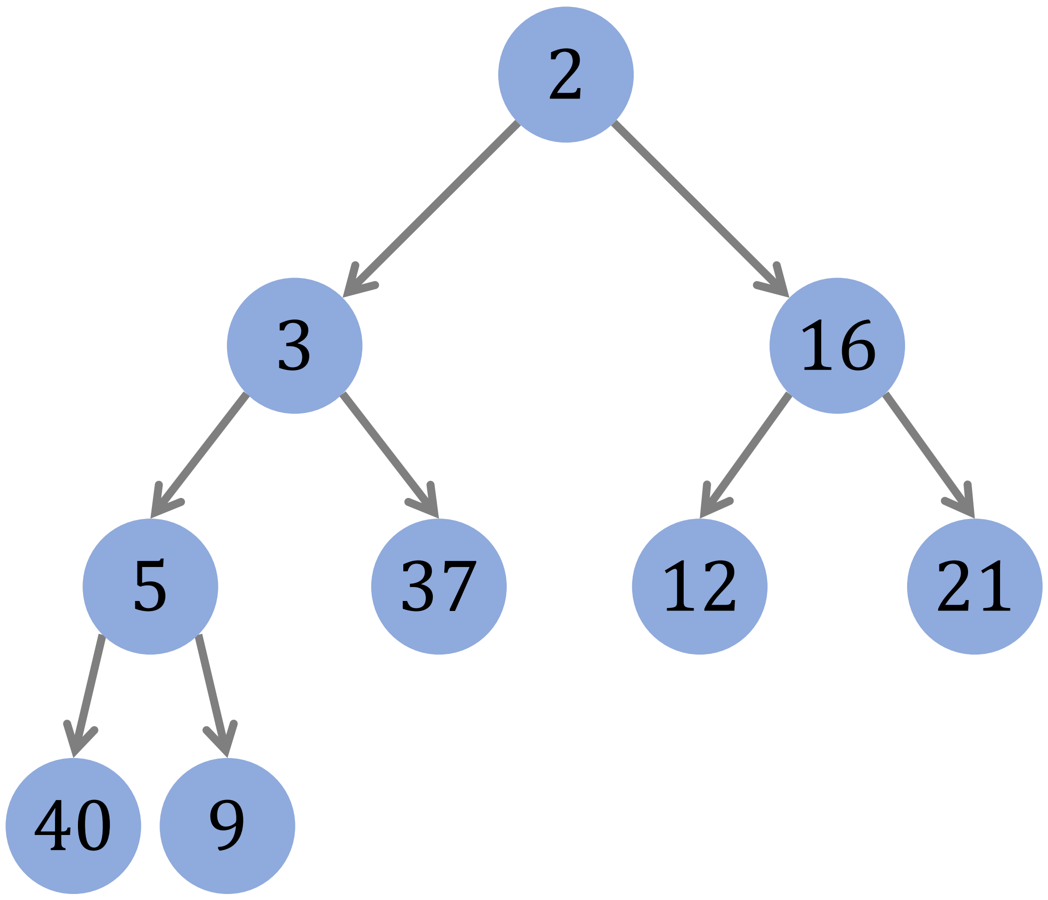


Is this a min-heap?

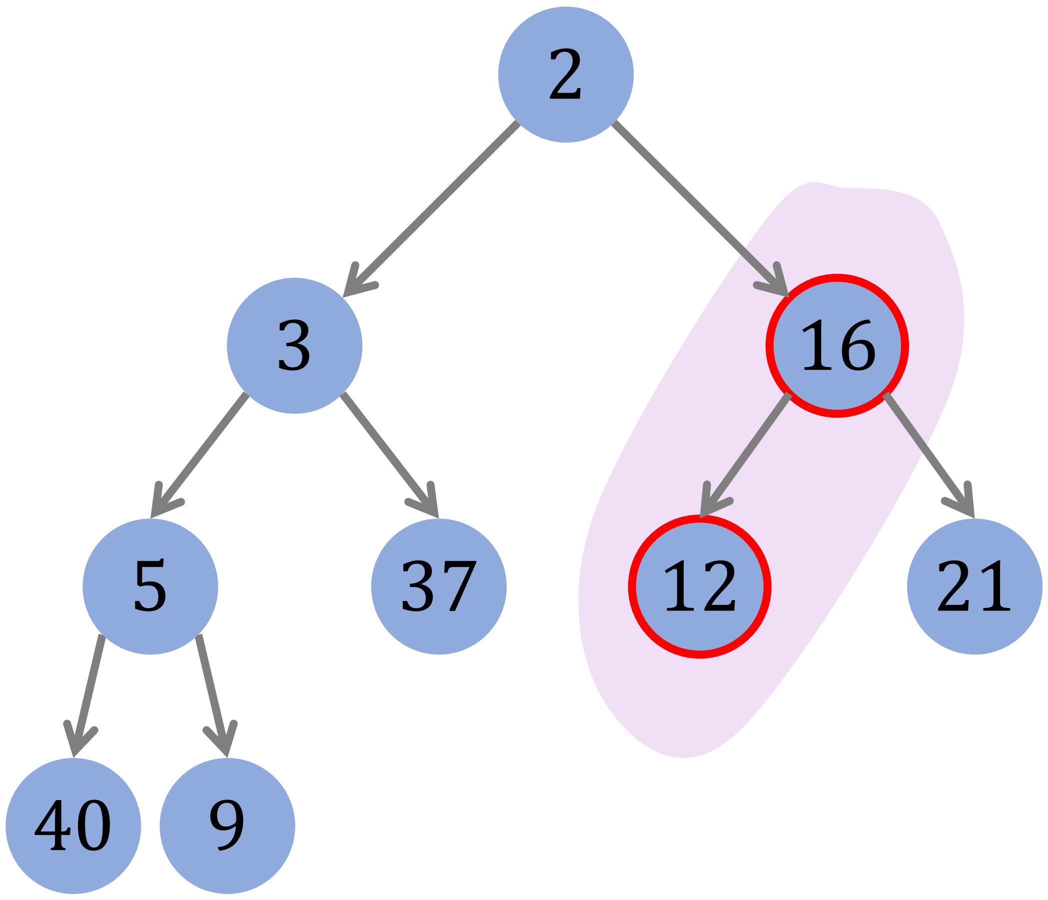


- **No!**
- It is not complete binary tree.
- This level is not filled.

Is this a min-heap?



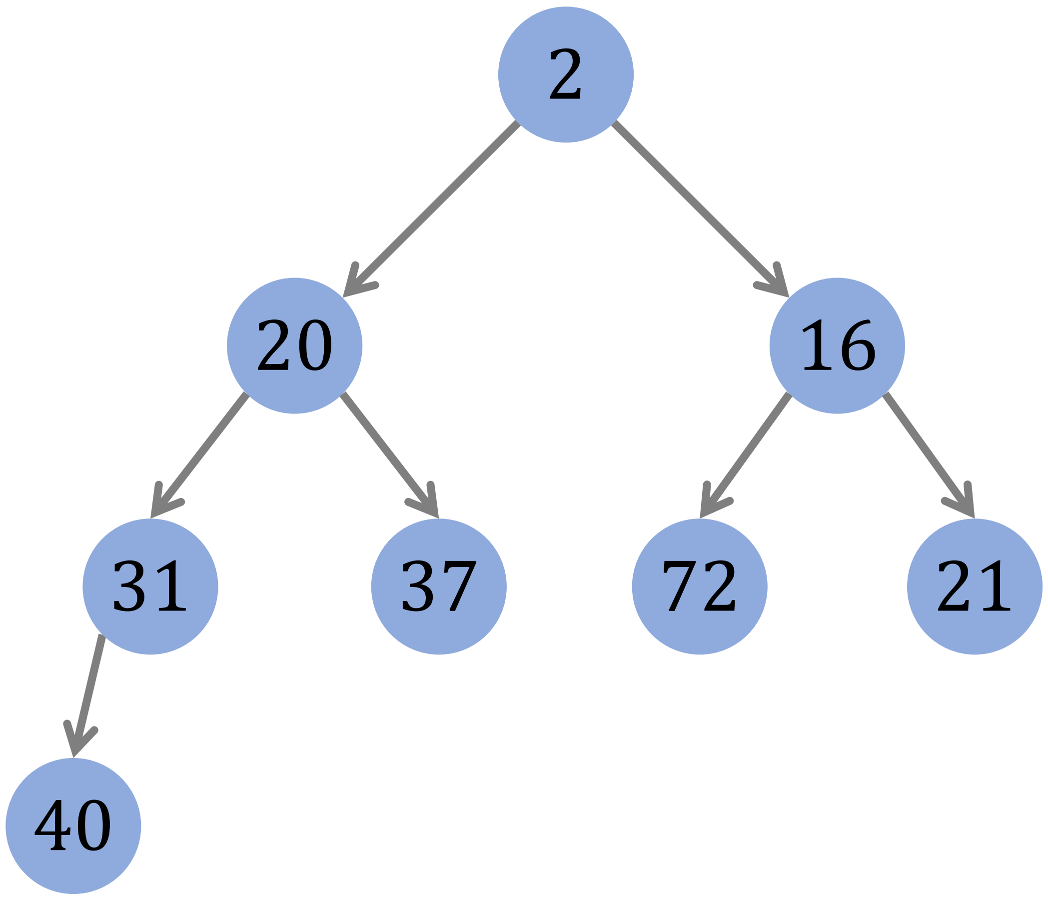
Is this a min-heap?



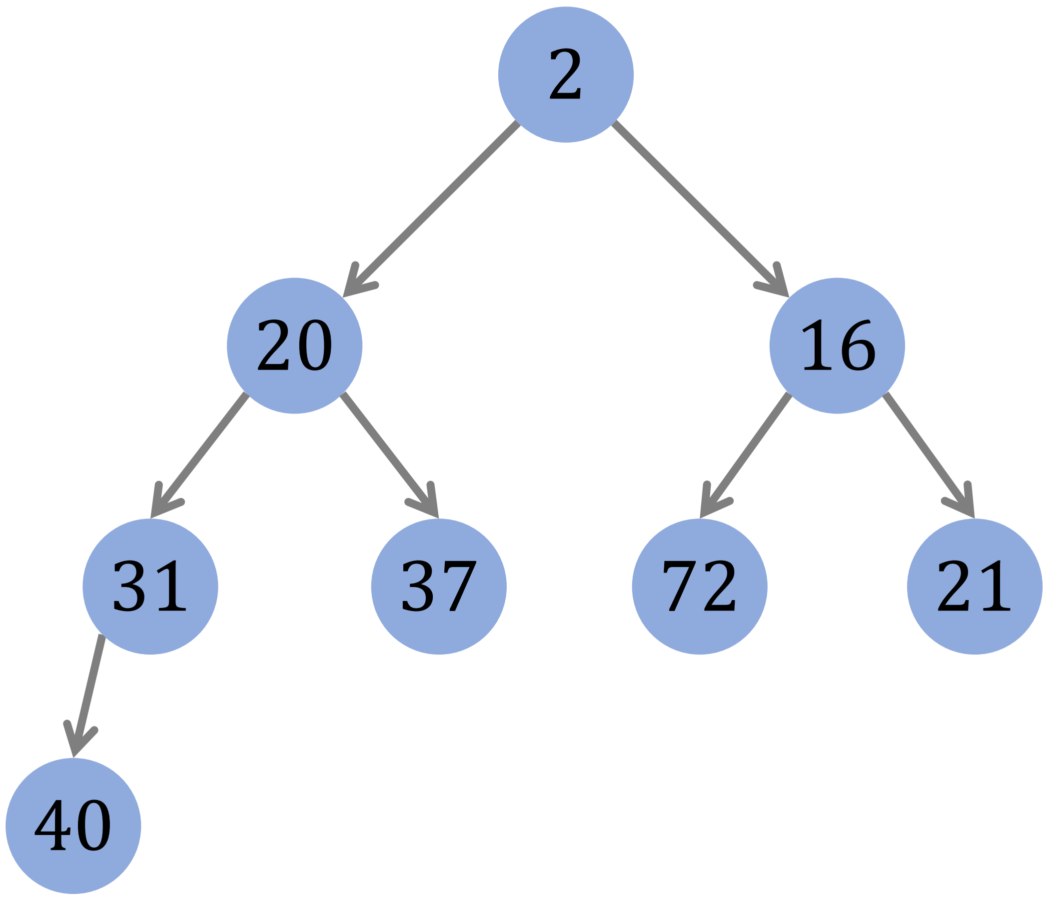
- **No!**
- The parent's key shouldn't be greater than the child's key.

Insert Vertices into Min-heap

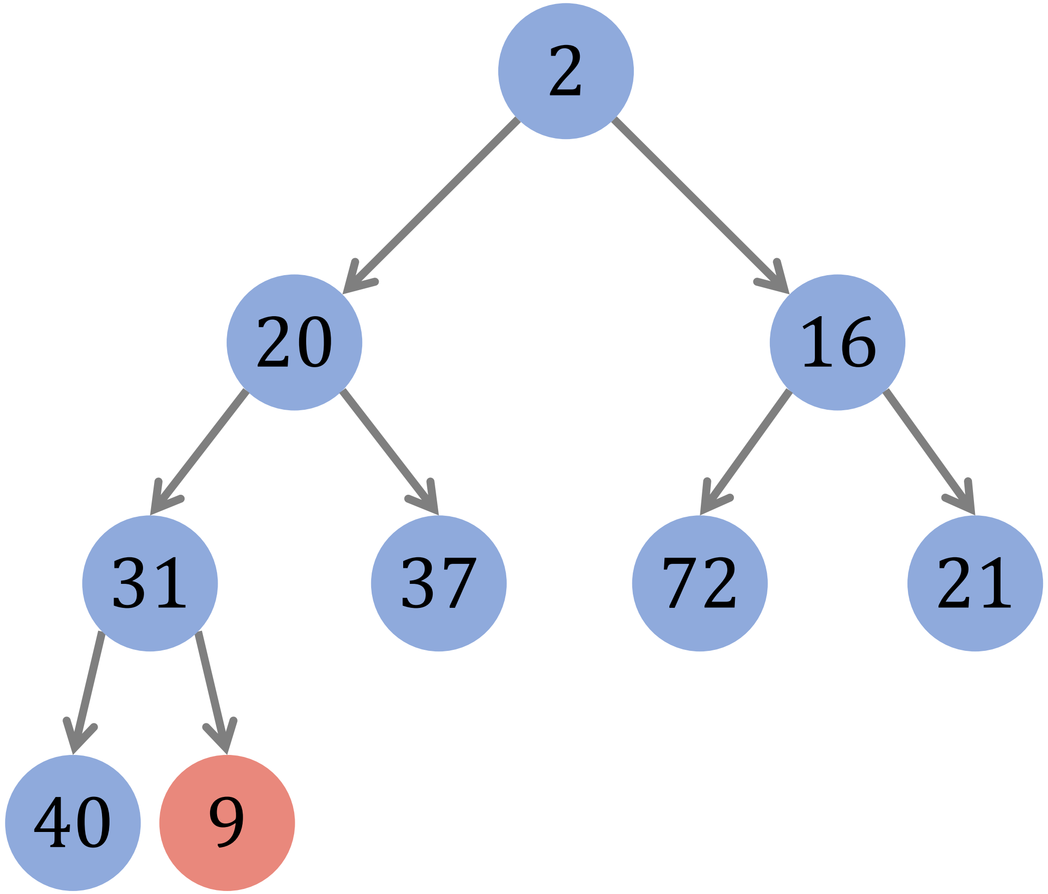
Current State



Insert(9)



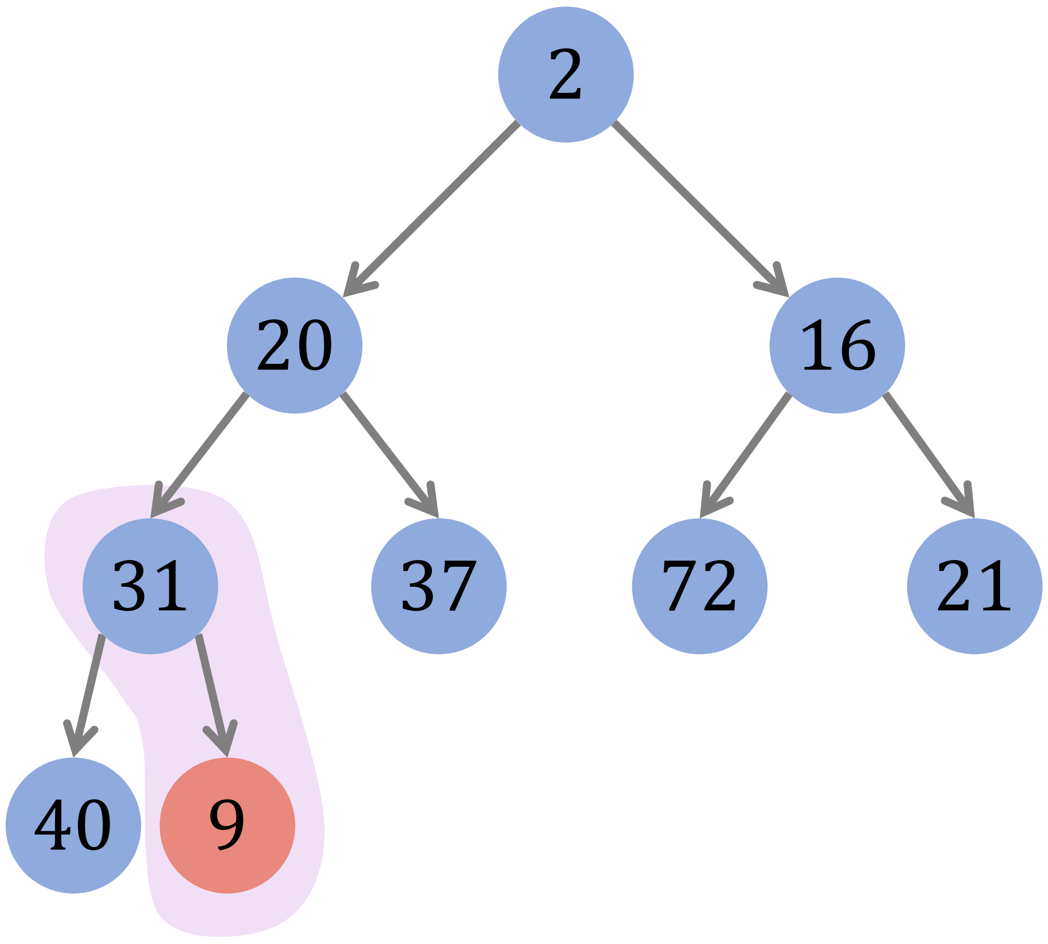
`Insert(9)`



Procedure

1. Insert the key at the end.
2. Percolate up.
   - Is the key is smaller than its parent?
   - If yes, then swap it and its parent.
   - If no, then stop.

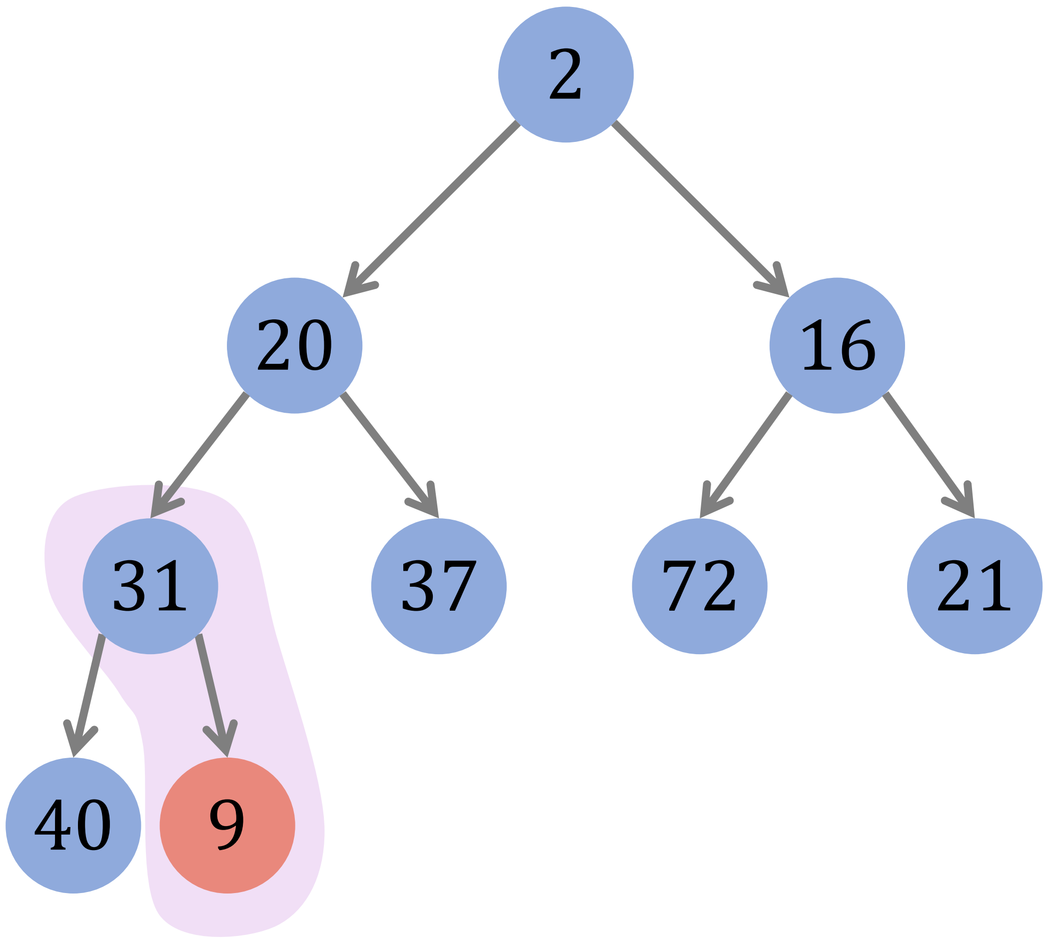
Insert(9)



Procedure

1. Insert the key at the end.
2. Percolate up.
   - Is the key is smaller than its parent?
   - If yes, then swap it and its parent.
   - If no, then stop.

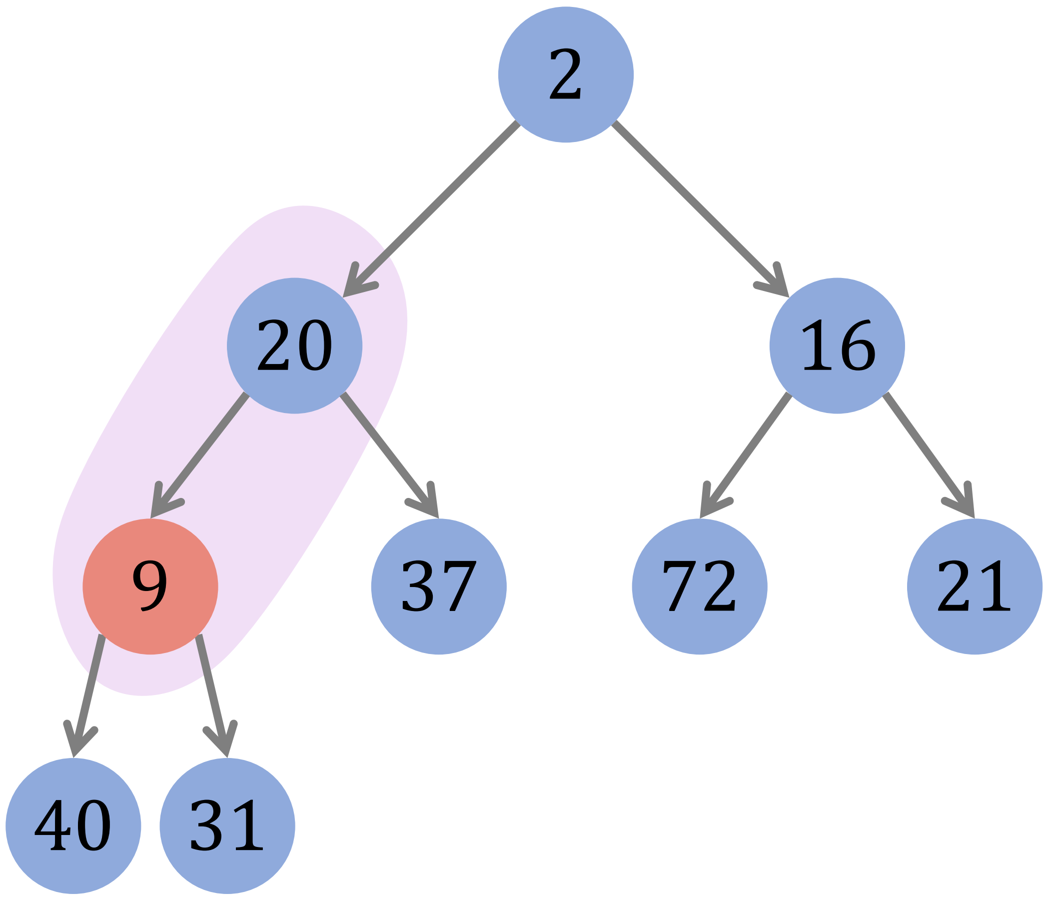
Insert(9)



Procedure

1. Insert the key at the end.
2. Percolate up.
   - Is the key is smaller than its parent?
   - If yes, then swap it and its parent.
   - If no, then stop.

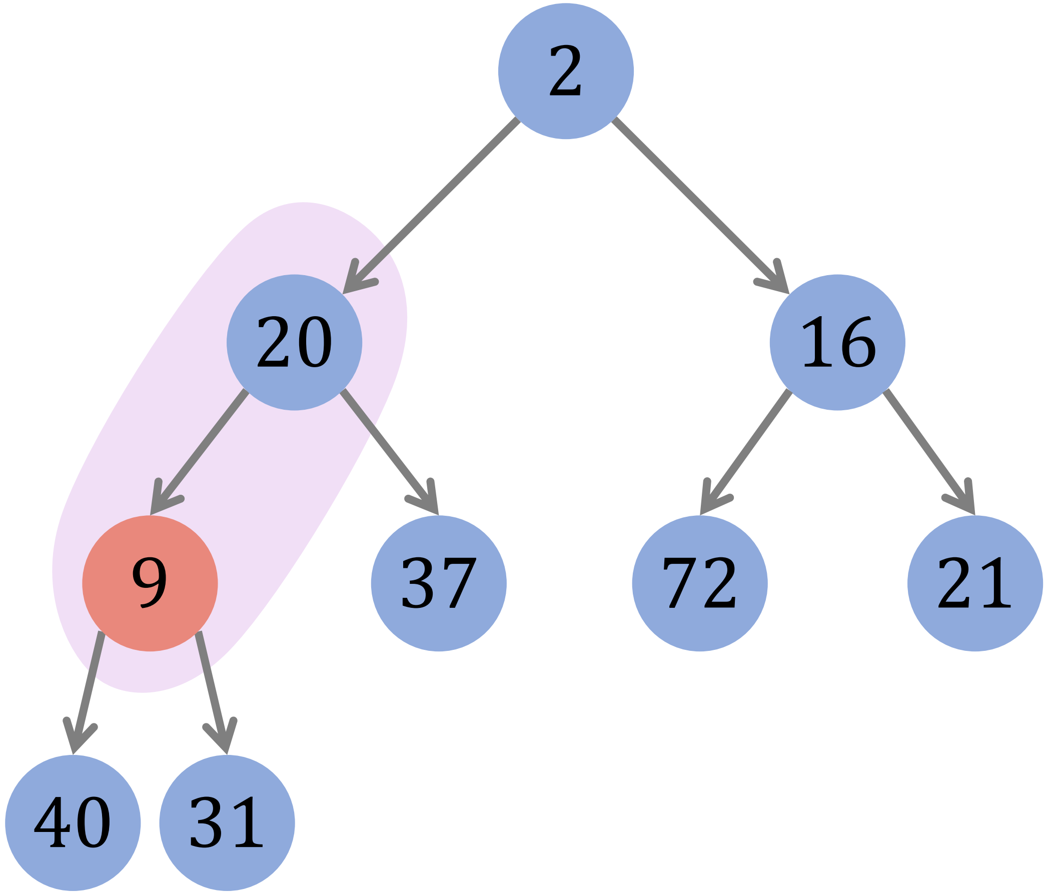
`Insert(9)`



Procedure

1. Insert the key at the end.
2. Percolate up.
   - Is the key is smaller than its parent?
   - If yes, then swap it and its parent.
   - If no, then stop.

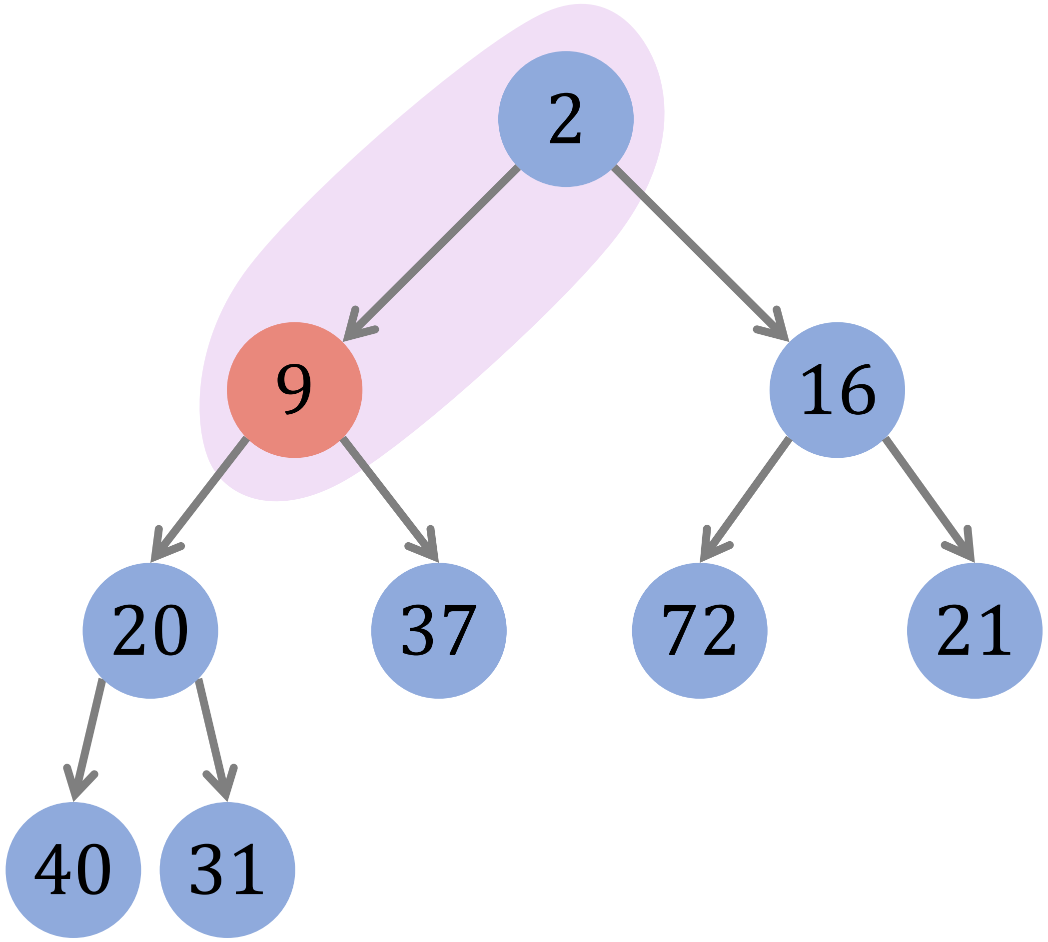
Insert(9)



Procedure

1. Insert the key at the end.
2. Percolate up.
   - Is the key is smaller than its parent?
   - If yes, then swap it and its parent.
   - If no, then stop.

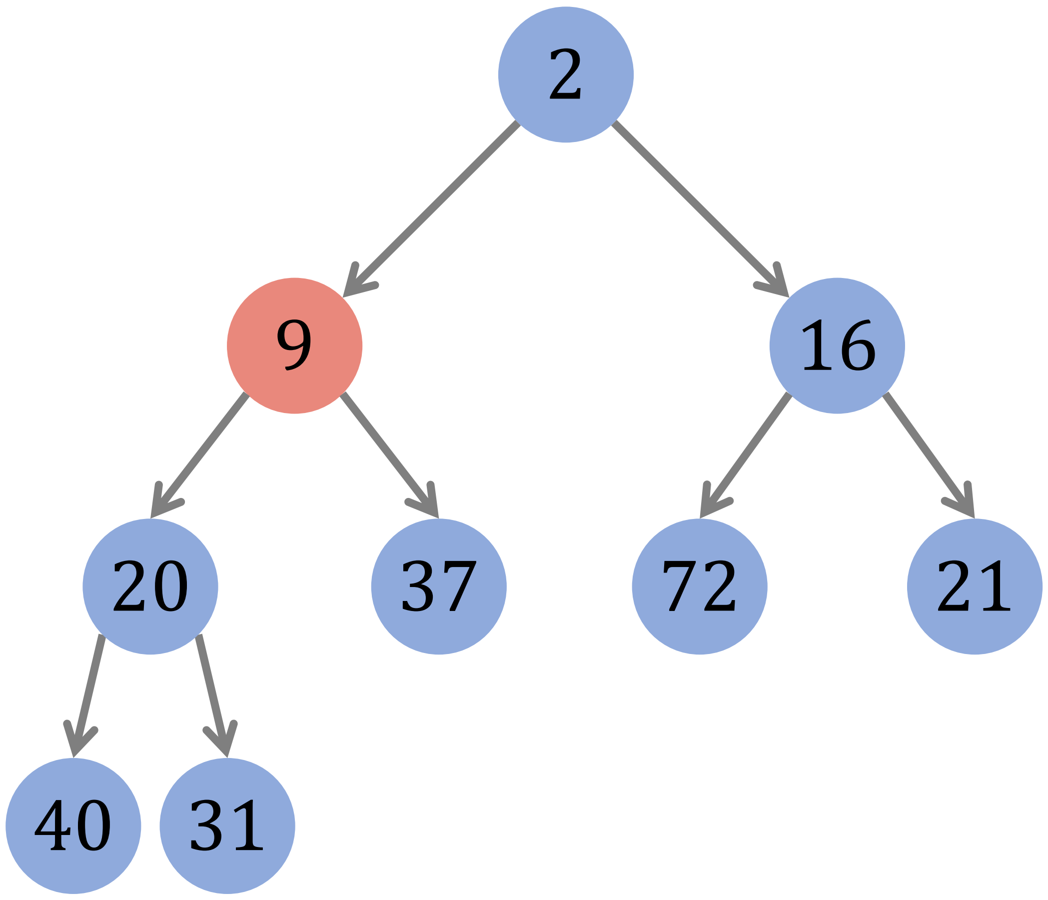
`Insert(9)`



Procedure

1. Insert the key at the end.
2. Percolate up.
    - Is the key is smaller than its parent?
    - If yes, then swap it and its parent.
    - If no, then stop.

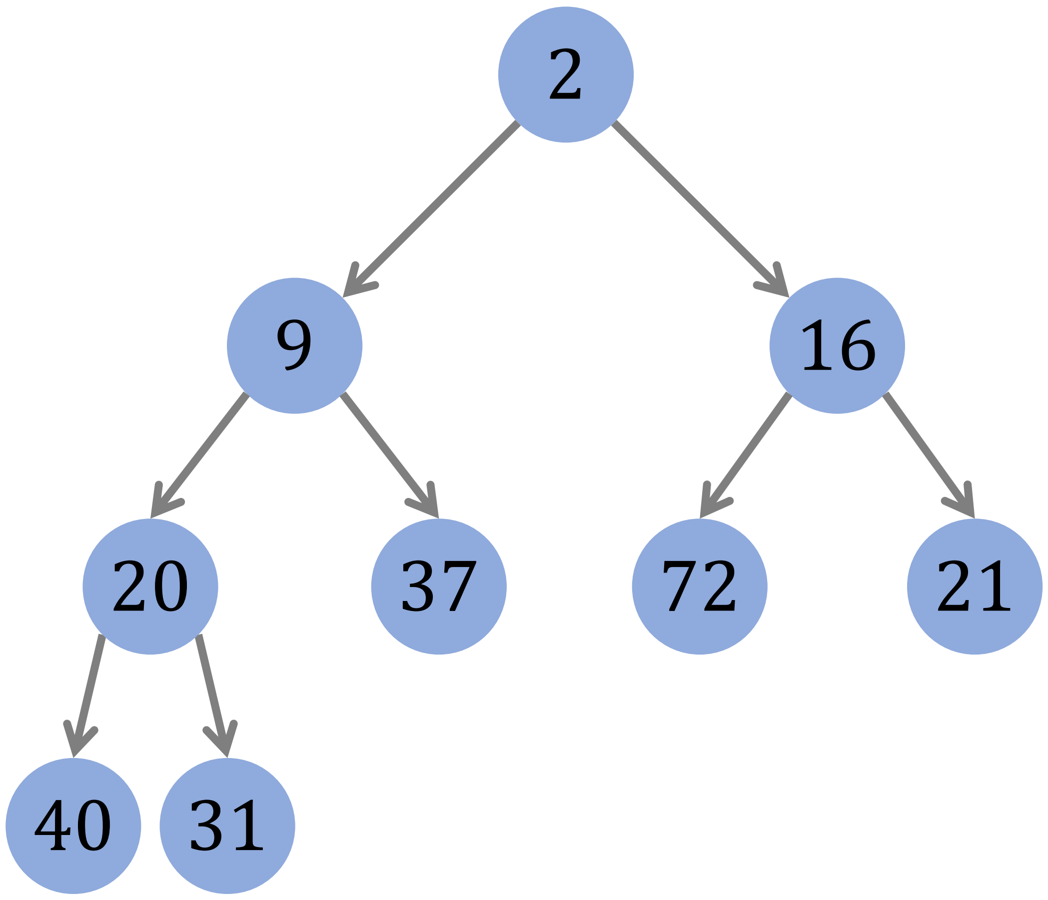
Insert(9)



Procedure

1. Insert the key at the end.
2. Percolate up.
   - Is the key is smaller than its parent?
   - If yes, then swap it and its parent.
   - If no, then stop.

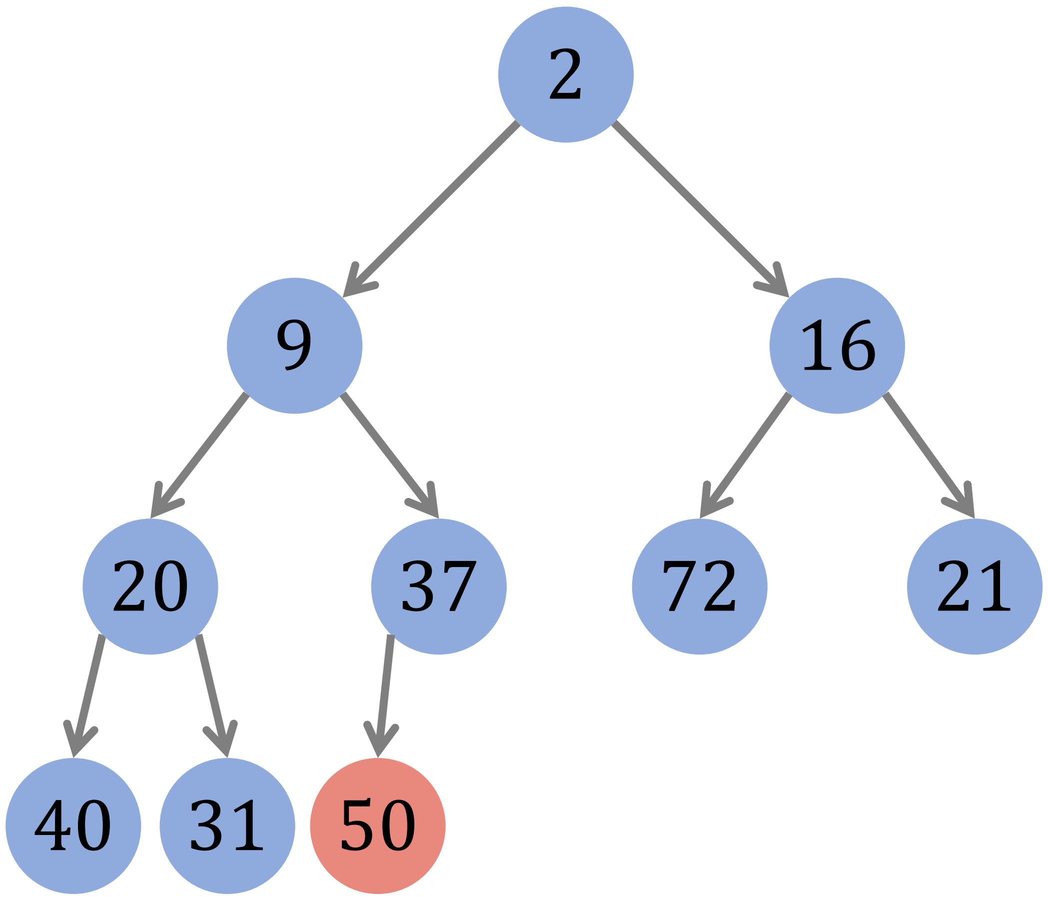
Current State



Procedure

1. Insert the key at the end.
2. Percolate up.
    - Is the key is smaller than its parent?
    - If yes, then swap it and its parent.
    - If no, then stop.

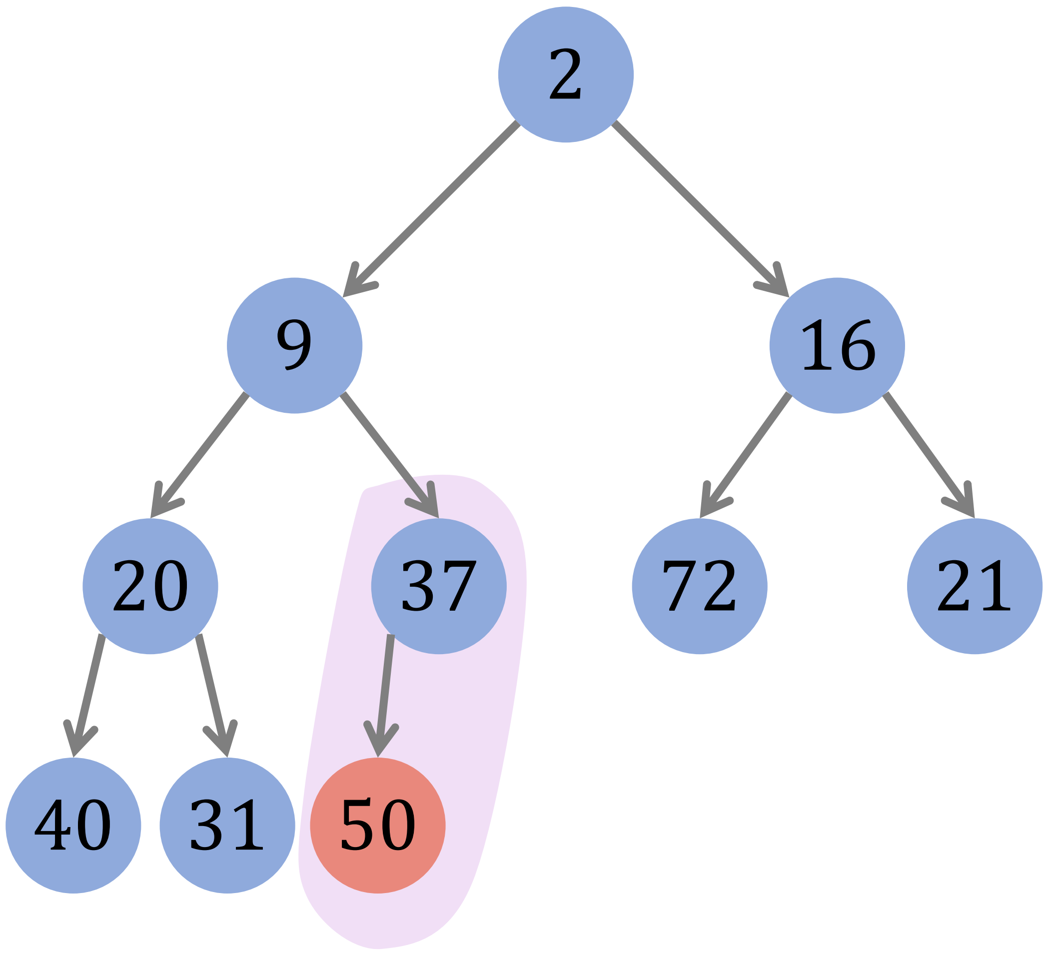
`Insert(50)`



Procedure

1. Insert the key at the end.
2. Percolate up.
   - Is the key is smaller than its parent?
   - If yes, then swap it and its parent.
   - If no, then stop.

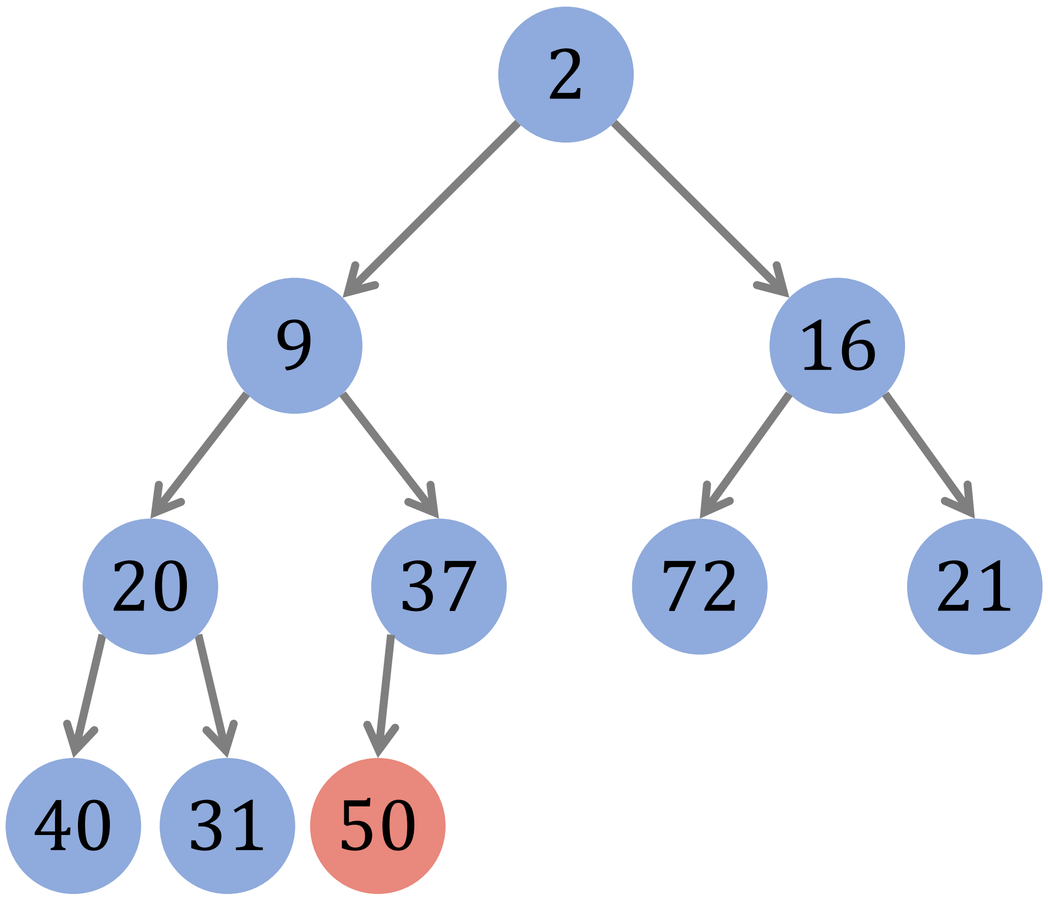
Insert(50)



Procedure

1. Insert the key at the end.
2. Percolate up.
   - Is the key is smaller than its parent?
   - If yes, then swap it and its parent.
   - If no, then stop.

Insert(50)

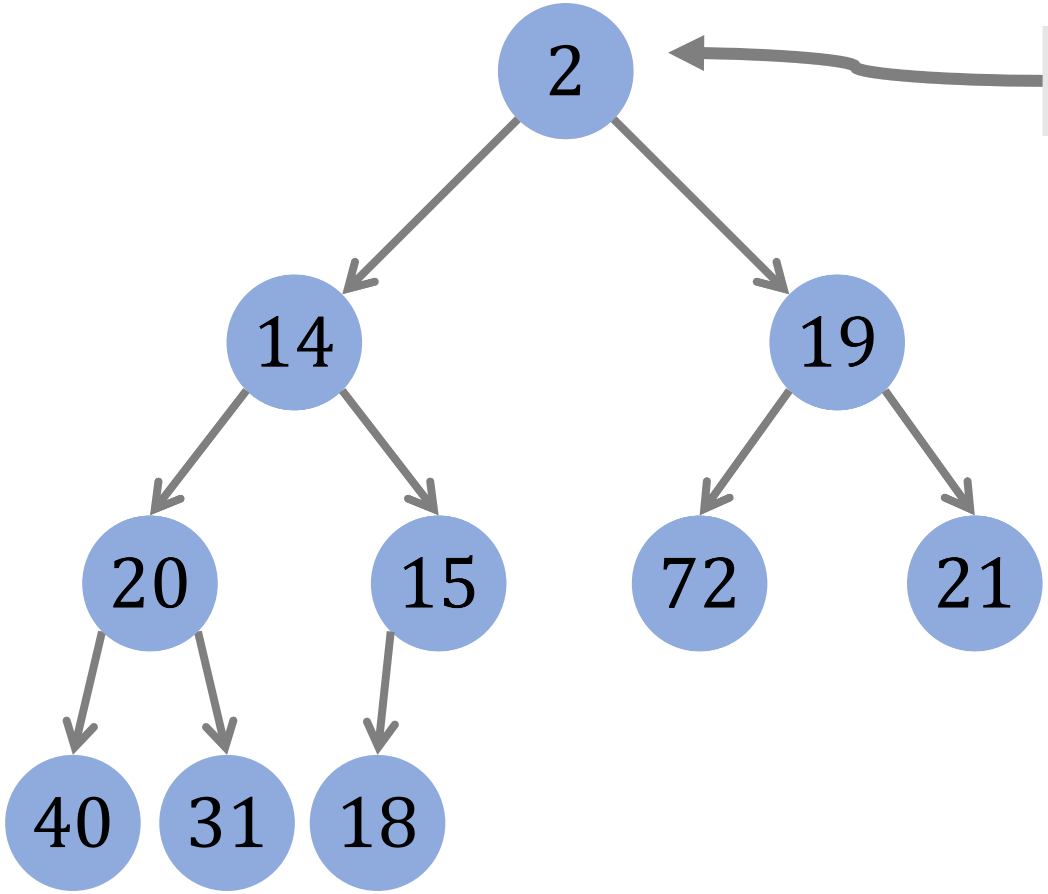


Procedure

1. Insert the key at the end.
2. Percolate up.
    - Is the key is smaller than its parent?
    - If yes, then swap it and its parent.
    - If no, then stop.

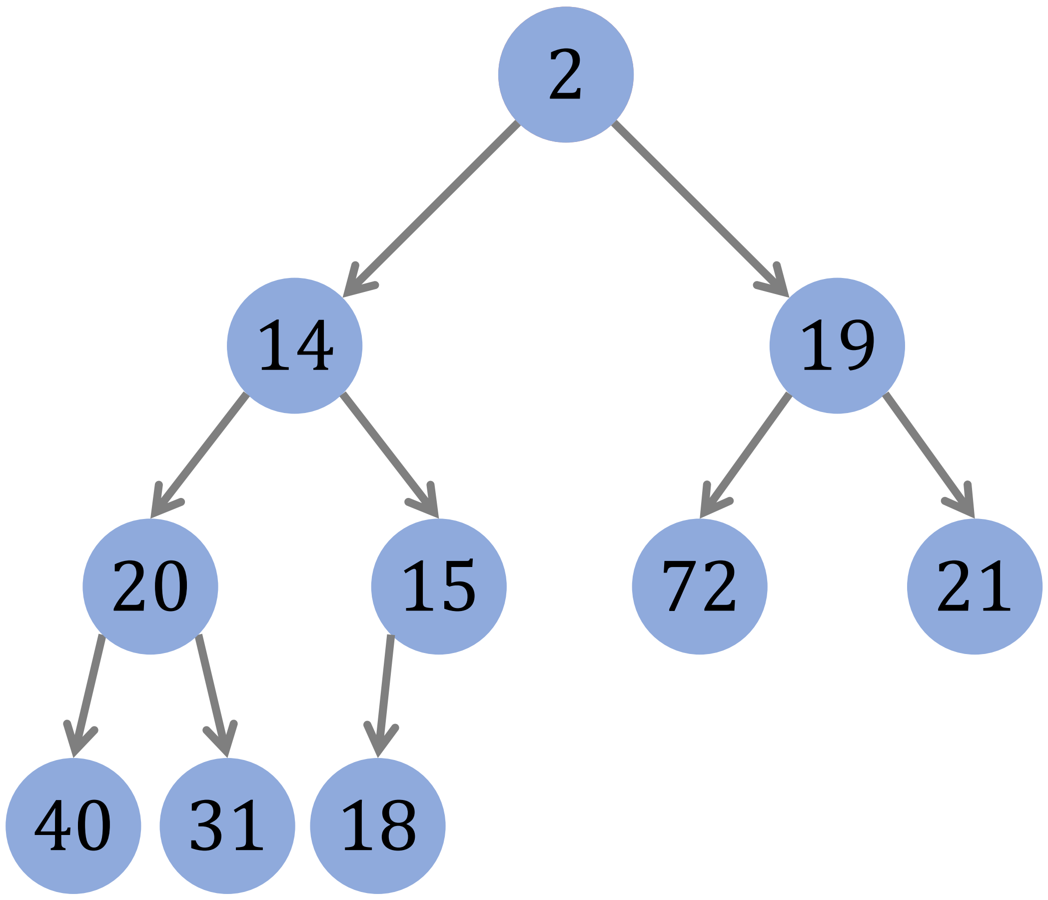
Delete Min from Min-heap

Current State



The root contains the minimum.

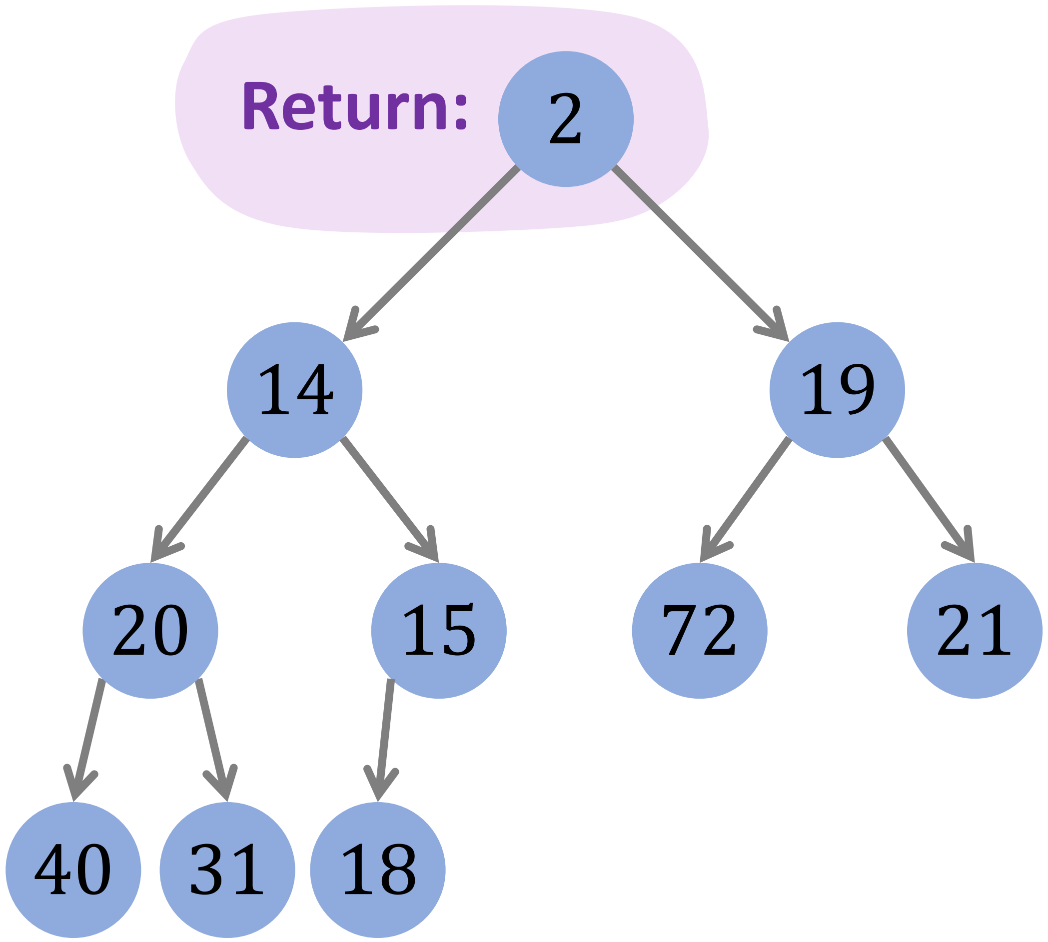
DeleteMin()



Procedure

1. Return and delete the root.
2. Move the last key to the root.
3. Percolate down.
   - Is the key bigger than a child?
   - If yes, swap it with the smaller child.
   - If no, then stop.

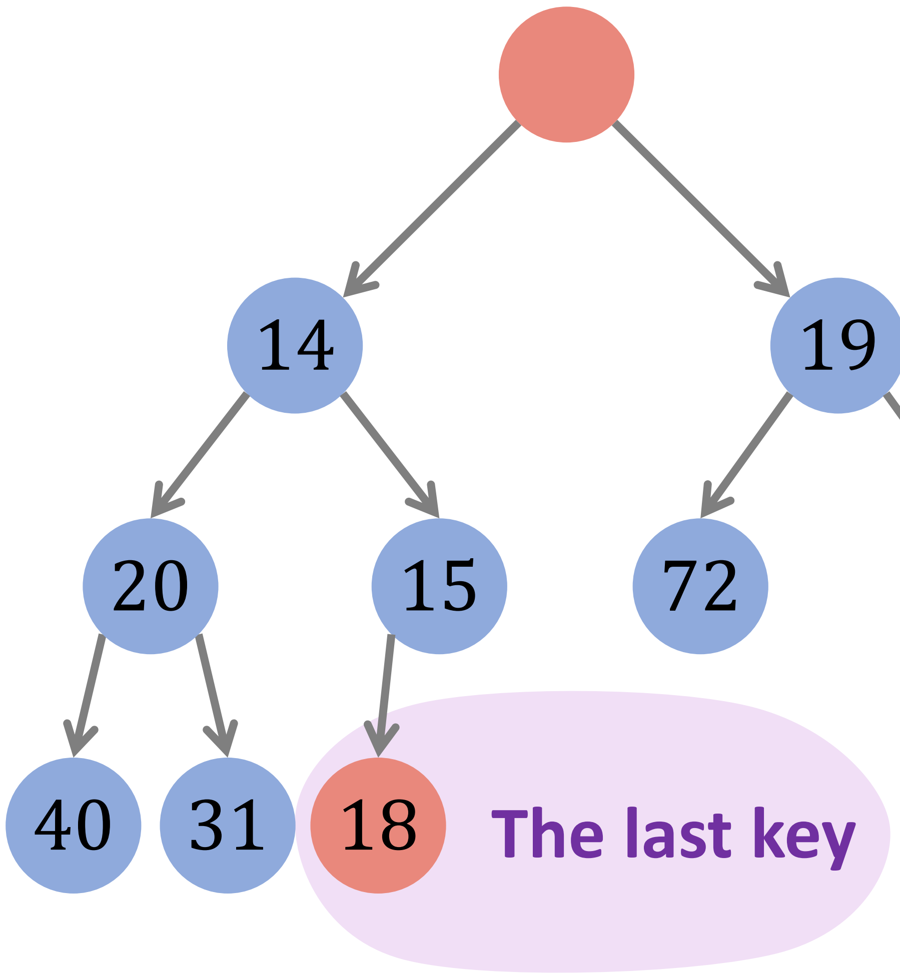
`DeleteMin()`



Procedure

1. Return and delete the root.
2. Move the last key to the root.
3. Percolate down.
    - Is the key bigger than a child?
    - If yes, swap it with the smaller child.
    - If no, then stop.

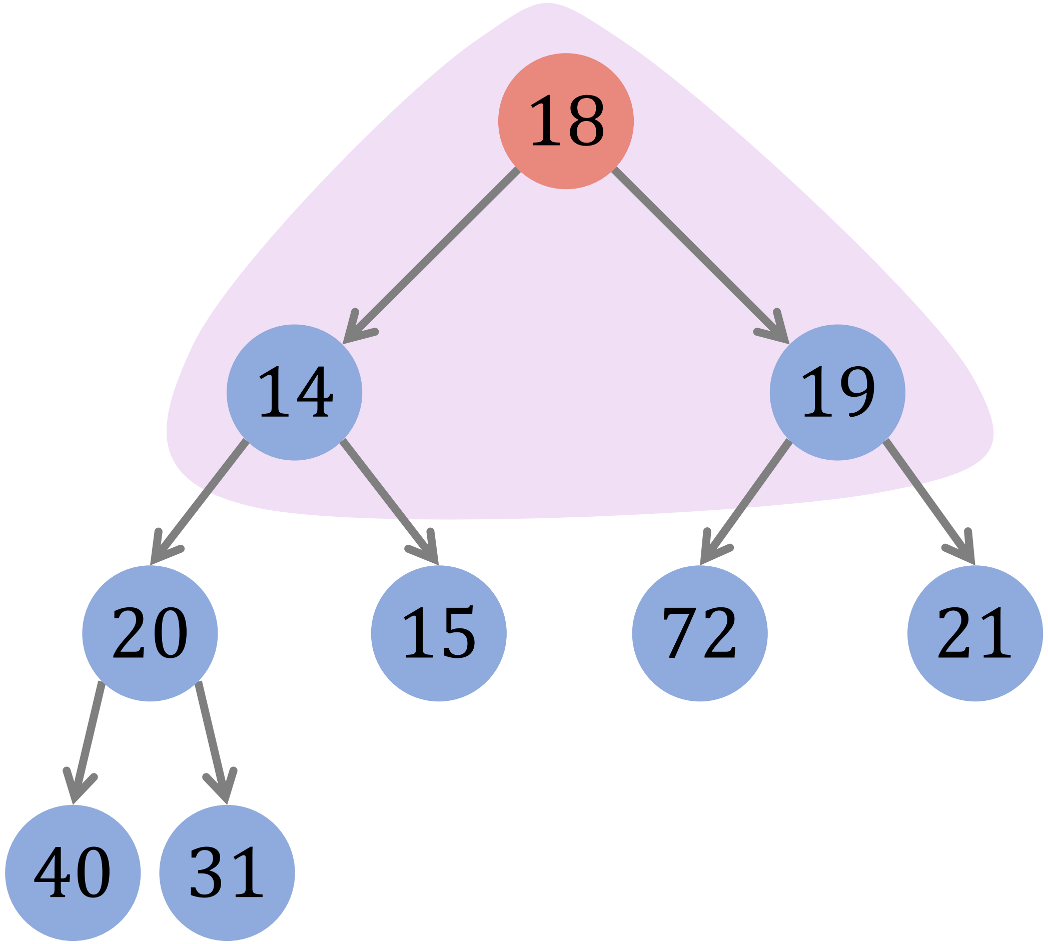
DeleteMin()



Procedure

1. Return and delete the root.
2. Move the last key to the root.
3. Percolate down.
   - Is the key bigger than a child?
   - If yes, swap it with the smaller child.
   - If no, then stop.

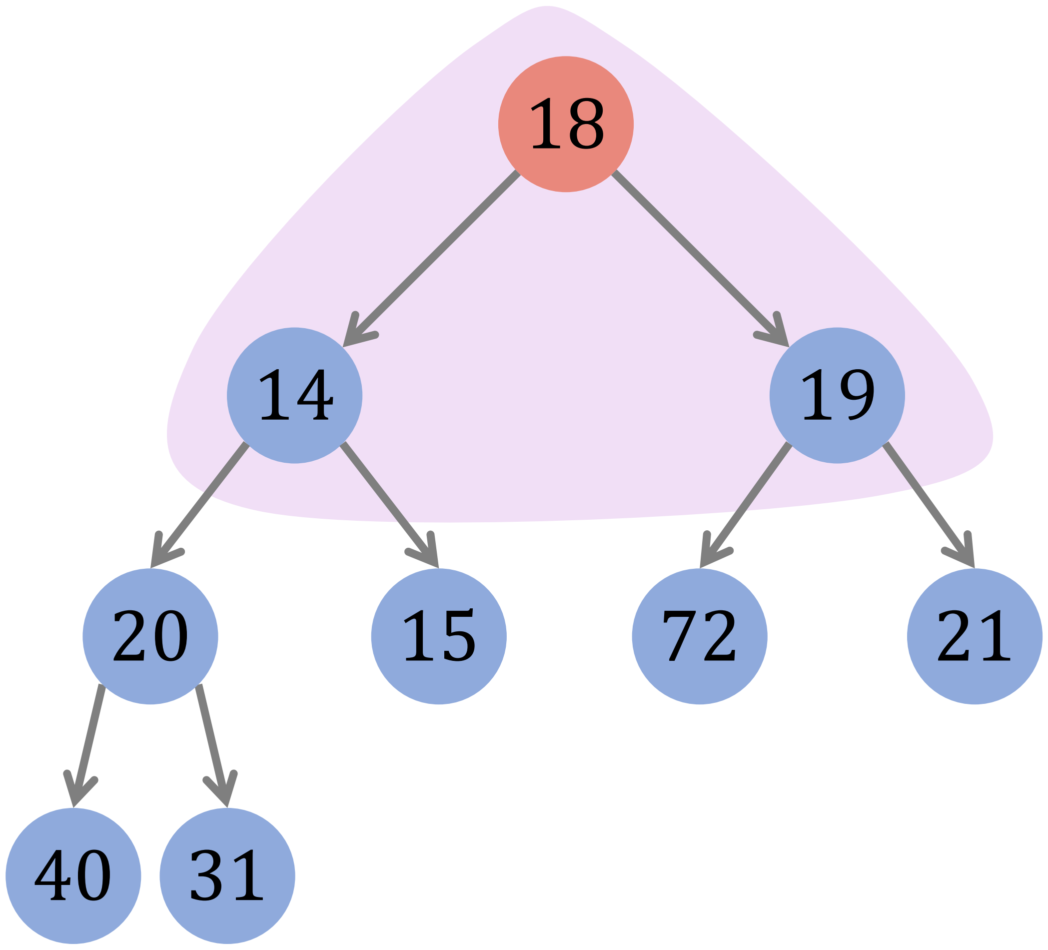
`DeleteMin()`



Procedure

1. Return and delete the root.
2. Move the last key to the root.
3. Percolate down.
   - Is the key bigger than a child?
   - If yes, swap it with the smaller child.
   - If no, then stop.

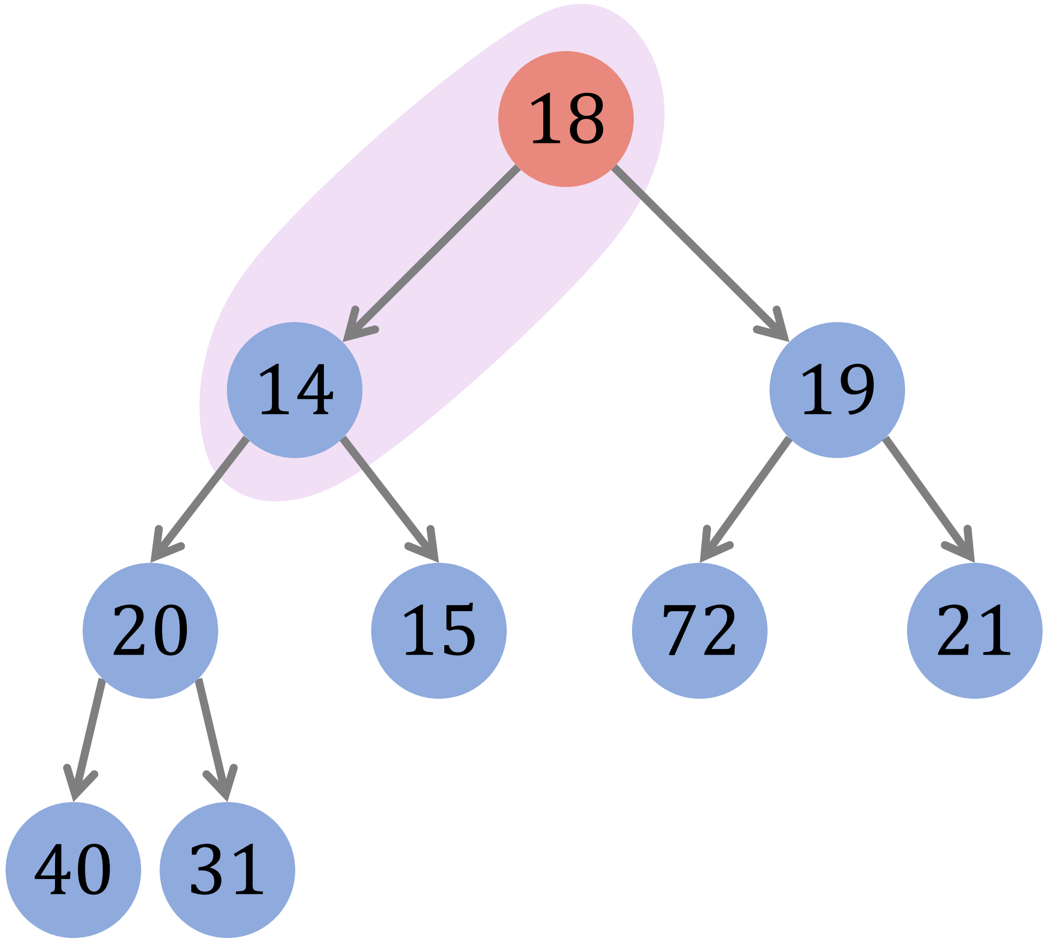
DeleteMin()



Procedure

1. Return and delete the root.
2. Move the last key to the root.
3. Percolate down.
   - Is the key bigger than a child?
   - If yes, swap it with the smaller child.
   - If no, then stop.

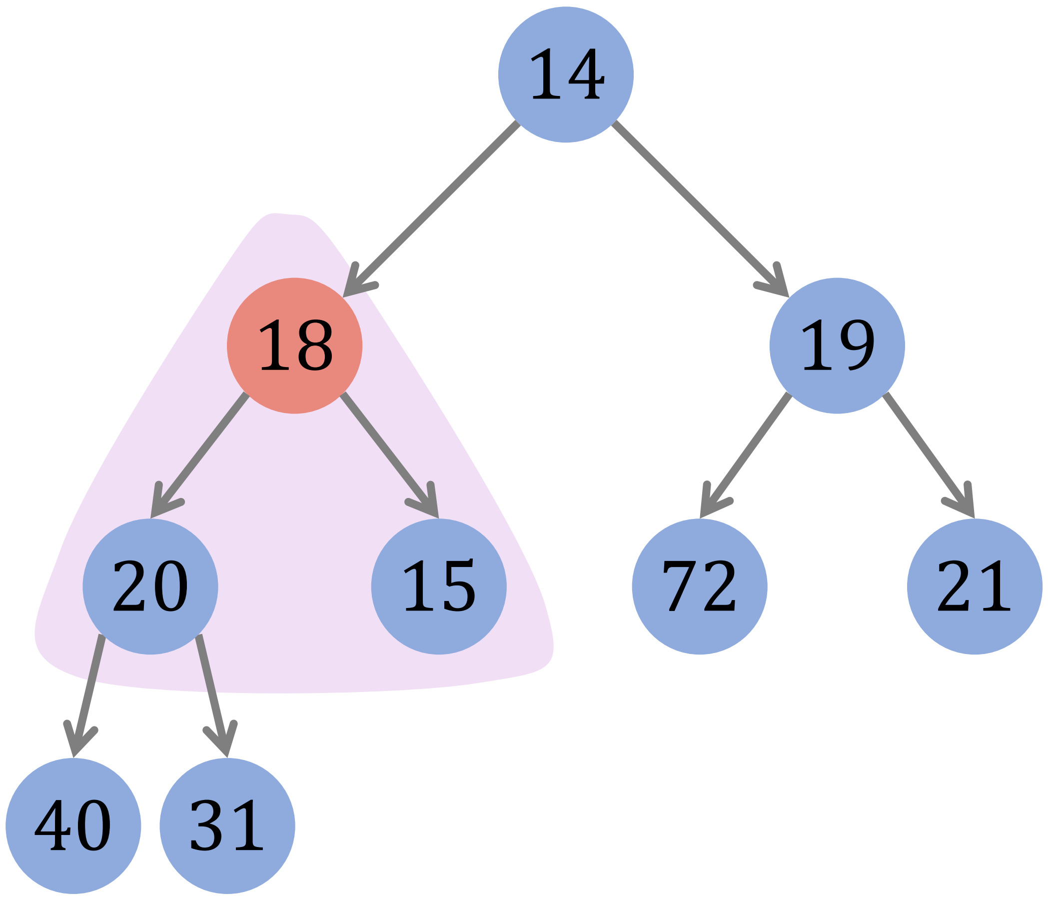
DeleteMin()



Procedure

1. Return and delete the root.
2. Move the last key to the root.
3. Percolate down.
    - Is the key bigger than a child?
    - If yes, swap it with the smaller child.
    - If no, then stop.

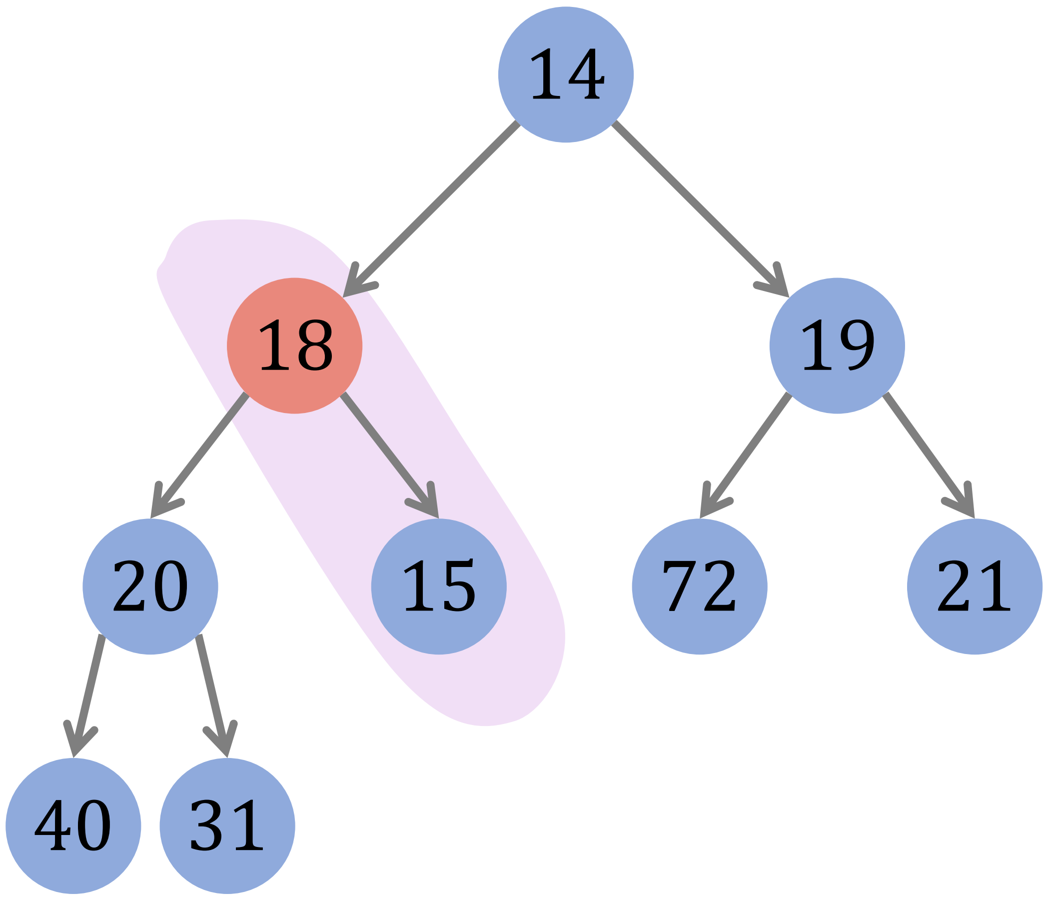
DeleteMin()



Procedure

1. Return and delete the root.
2. Move the last key to the root.
3. Percolate down.
   - Is the key bigger than a child?
   - If yes, swap it with the smaller child.
   - If no, then stop.

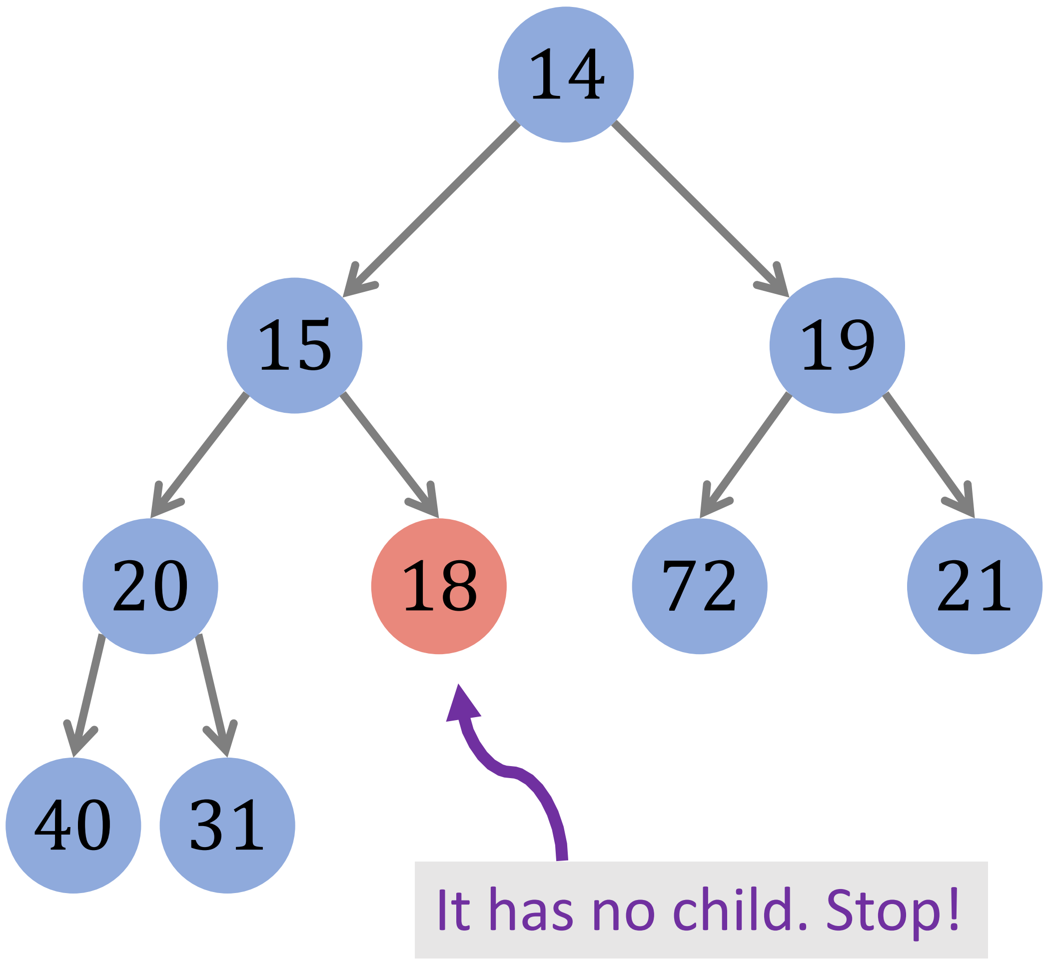
DeleteMin()



Procedure

1. Return and delete the root.
2. Move the last key to the root.
3. Percolate down.
   - Is the key bigger than a child?
   - If yes, swap it with the smaller child.
   - If no, then stop.

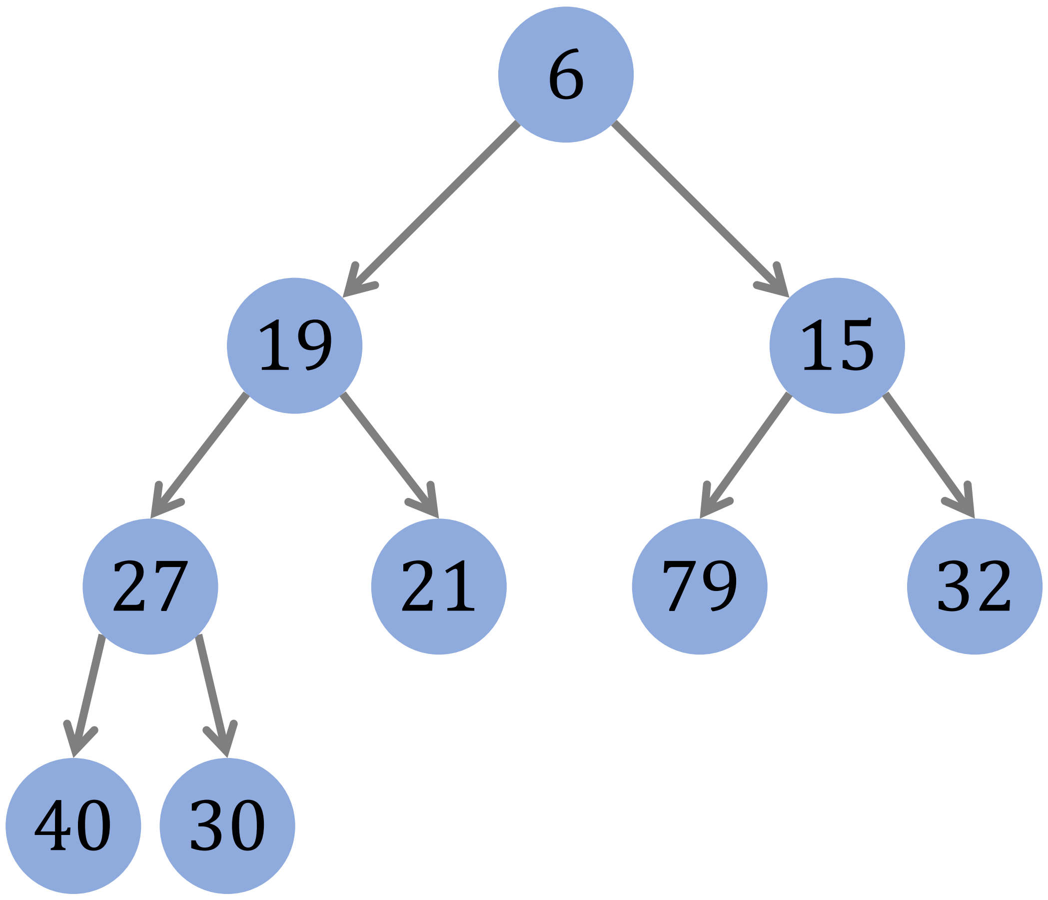
`DeleteMin()`



Procedure

1. Return and delete the root.
2. Move the last key to the root.
3. Percolate down.
   - Is the key bigger than a child?
   - If yes, swap it with the smaller child.
   - If no, then stop.

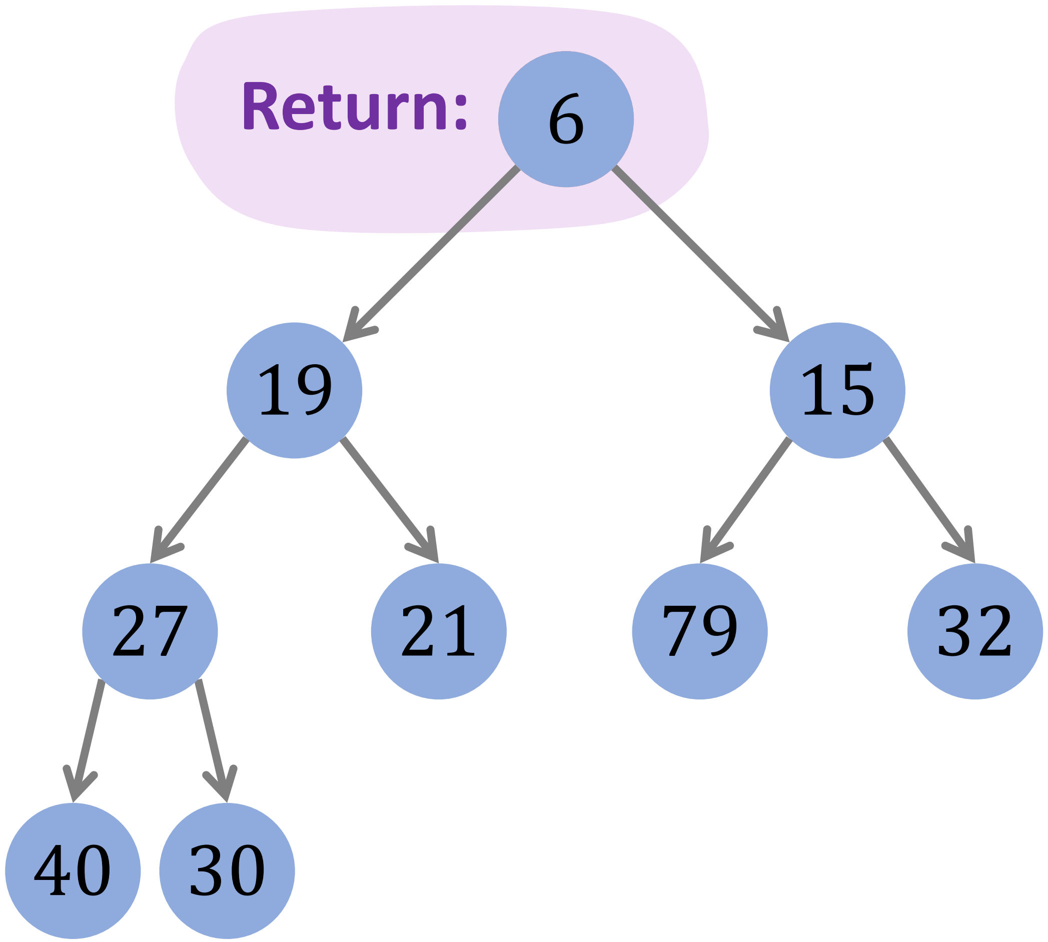
Current State



Procedure

1. Return and delete the root.
2. Move the last key to the root.
3. Percolate down.
   - Is the key bigger than a child?
   - If yes, swap it with the smaller child.
   - If no, then stop.

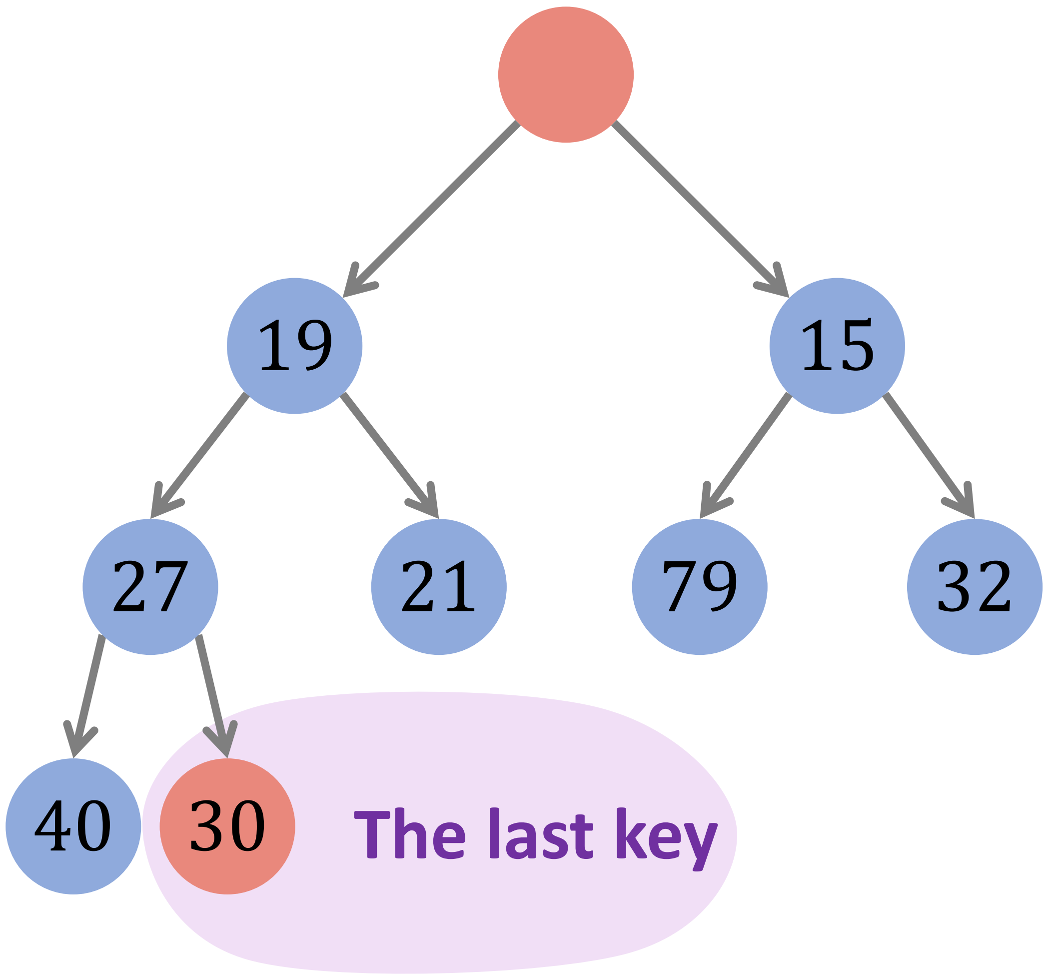
DeleteMin()



Procedure

1. Return and delete the root.
2. Move the last key to the root.
3. Percolate down.
    - Is the key bigger than a child?
    - If yes, swap it with the smaller child.
    - If no, then stop.

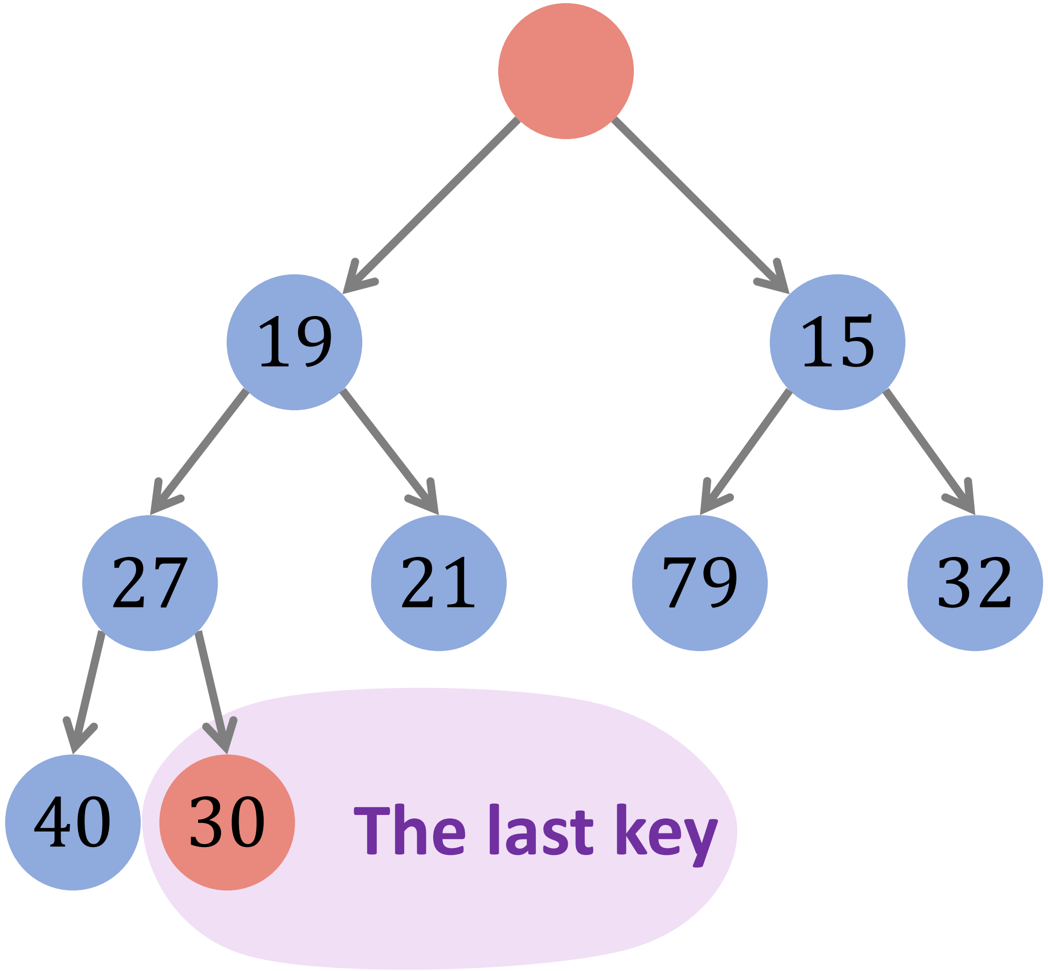
`DeleteMin()`



Procedure

1. Return and delete the root.
2. Move the last key to the root.
3. Percolate down.
   - Is the key bigger than a child?
   - If yes, swap it with the smaller child.
   - If no, then stop.

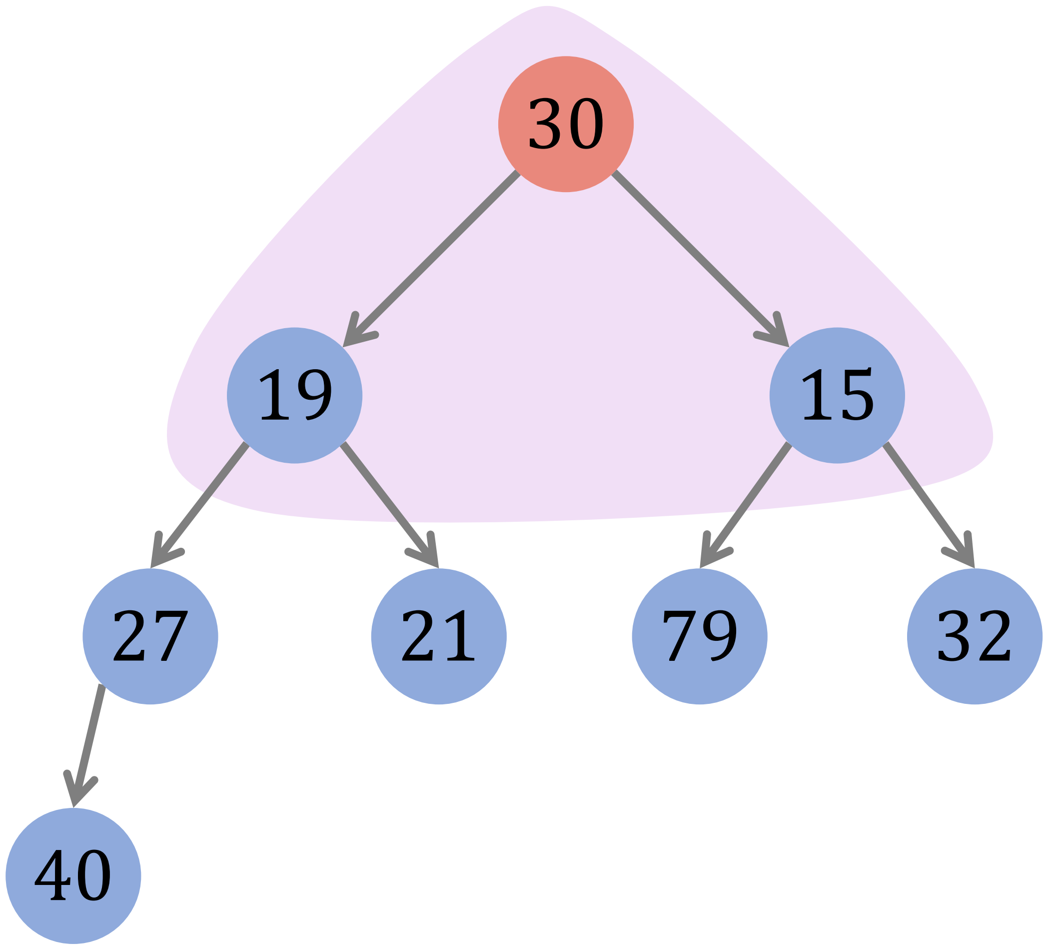
`DeleteMin()`



Procedure

1. Return and delete the root.
2. Move the last key to the root.
3. Percolate down.
   - Is the key bigger than a child?
   - If yes, swap it with the smaller child.
   - If no, then stop.

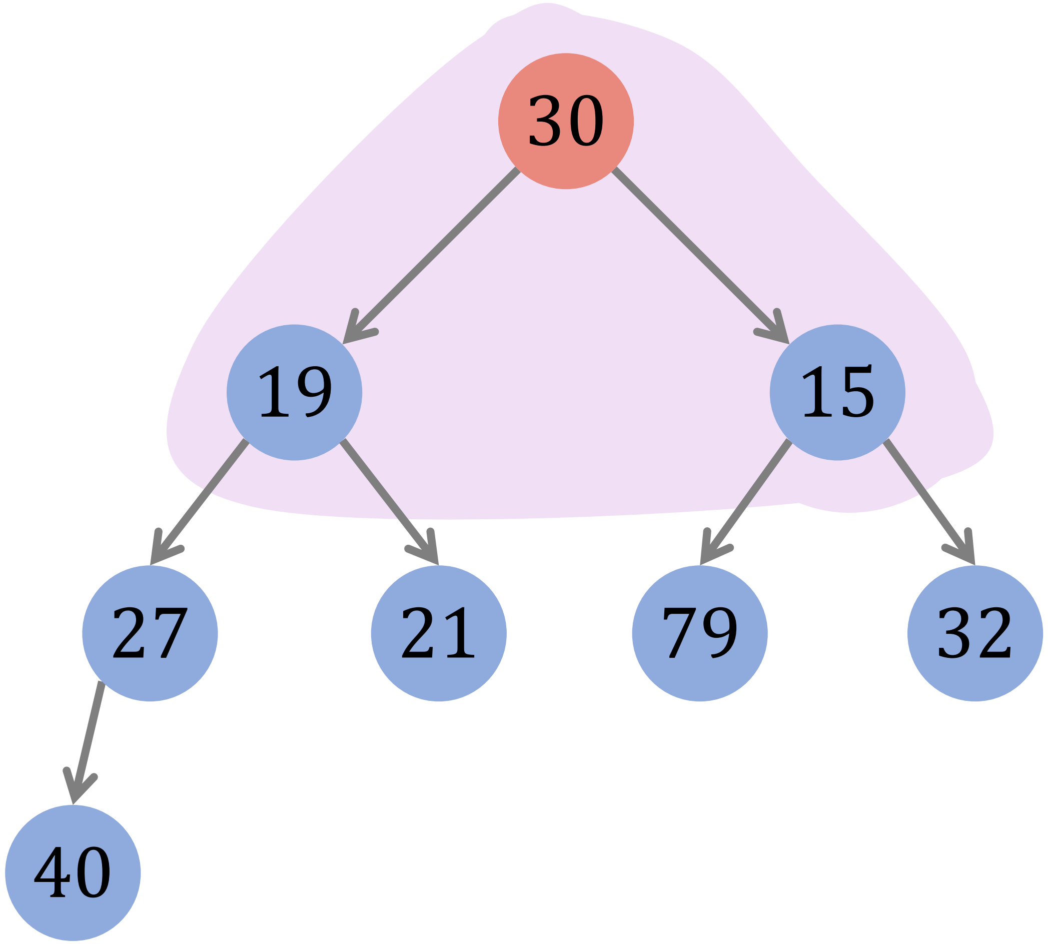
DeleteMin()



Procedure

1. Return and delete the root.
2. Move the last key to the root.
3. Percolate down.
    - Is the key bigger than a child?
    - If yes, swap it with the smaller child.
    - If no, then stop.

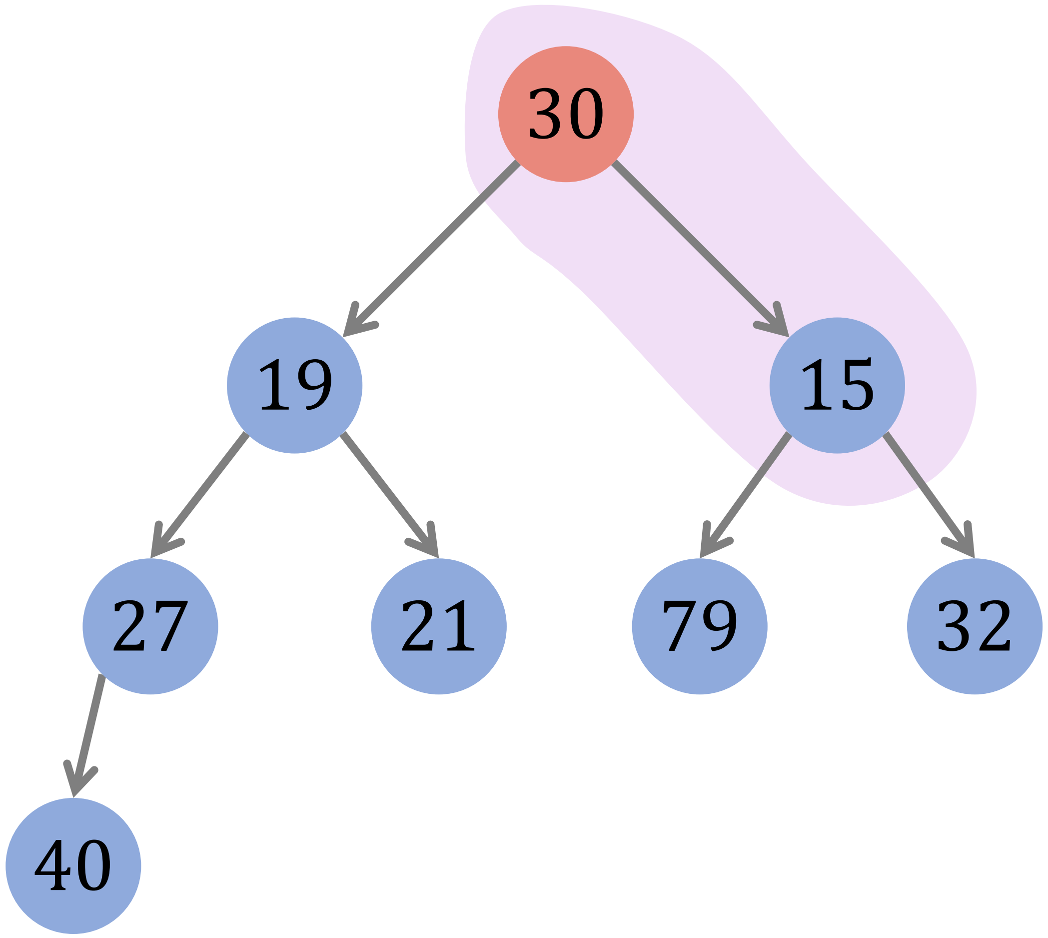
`DeleteMin()`



Procedure

1. Return and delete the root.
2. Move the last key to the root.
3. Percolate down.
    - Is the key bigger than a child?
    - If yes, swap it with the smaller child.
    - If no, then stop.

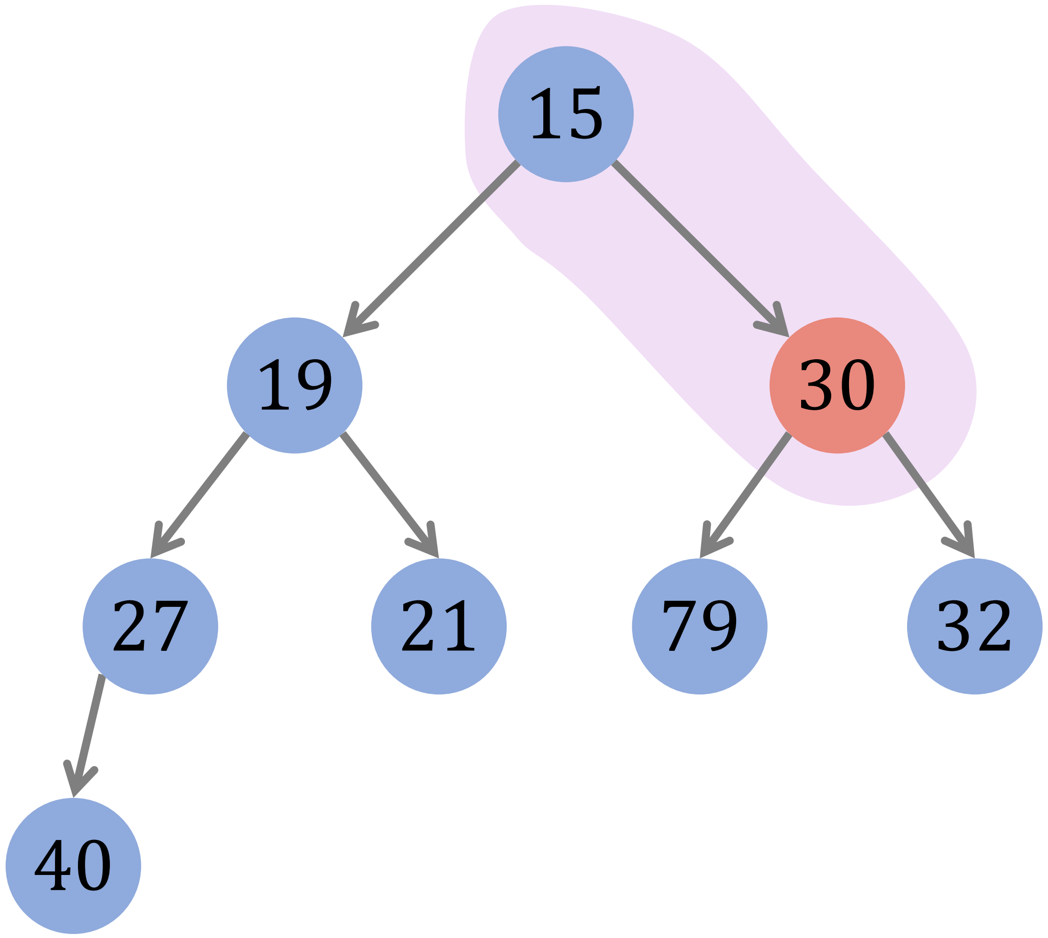
`DeleteMin()`



Procedure

1. Return and delete the root.
2. Move the last key to the root.
3. Percolate down.
    - Is the key bigger than a child?
    - If yes, swap it with the smaller child.
    - If no, then stop.

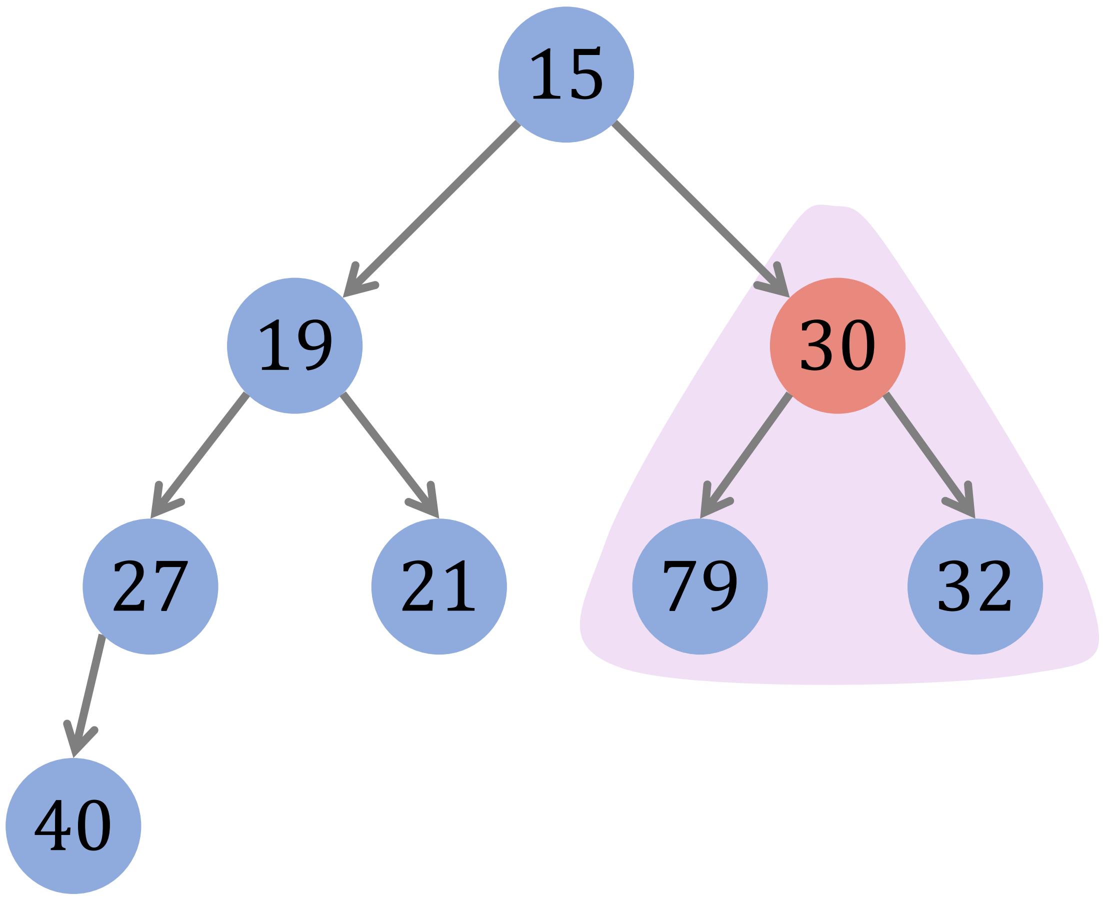
`DeleteMin()`



Procedure

1. Return and delete the root.
2. Move the last key to the root.
3. Percolate down.
   - Is the key bigger than a child?
   - If yes, swap it with the smaller child.
   - If no, then stop.

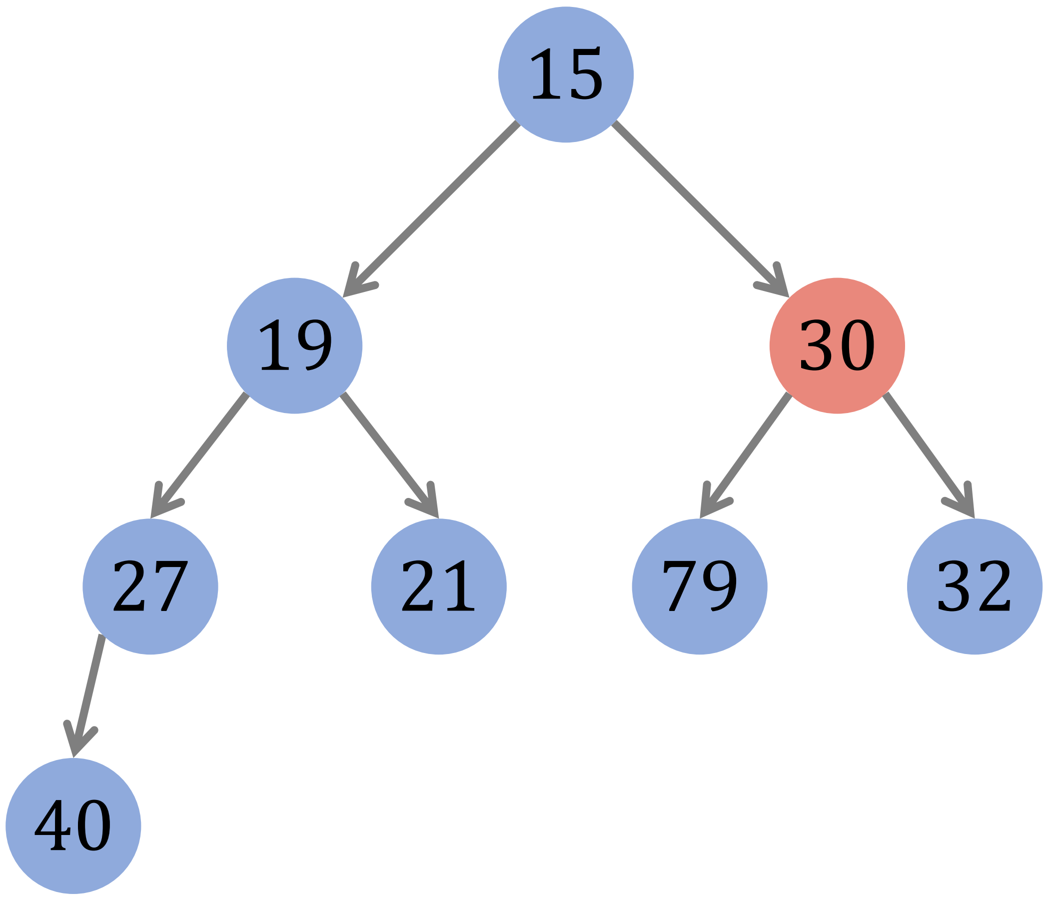
DeleteMin()



Procedure

1. Return and delete the root.
2. Move the last key to the root.
3. Percolate down.
   - Is the key bigger than a child?
   - If yes, swap it with the smaller child.
   - If no, then stop.

`DeleteMin()`



Procedure

1. Return and delete the root.
2. Move the last key to the root.
3. Percolate down.
    - Is the key bigger than a child?
    - If yes, swap it with the smaller child.
    - If no, then stop.

Summary

Summary

- Min-heap is a complete binary tree.
- Thus, min-heap can be implemented using an array.
- In a min-heap, a parent’s key must be less than or equal to its children’s keys.
- Min-heap is a kind of priority queue.

Time Complexities

- Let n be the number of vertices.
- The depth of the min-heap is $\log_2 n$.
- **`insert(i)`**: $O(\log n)$ time.
- **`deleteMin()`**: $O(\log n)$ time.

Questions

Q1: Are these complete binary trees min-heaps?

Tree 1:

15	18	19	20	31	72	21	40				

Tree 2:

9	60	14	72	66	22	56	92	88	68	69	24

Tree 3:

7	19	26	36	22	23	42	42	55	23		

Q2: After insert(20), what will the min-heaps be?

Tree 1:

4	18	12	24	31	72	21	40	26	32		

Tree 2:

9	50	14	76	66	22	43	92	88	68	69	

Tree 3:

7	19	26	36	22	98	42	43	55	23		

Q3: After deleteMin(), what will the min-heaps be?

Tree 1:

4	18	12	20	31	72	21	40	26	32		

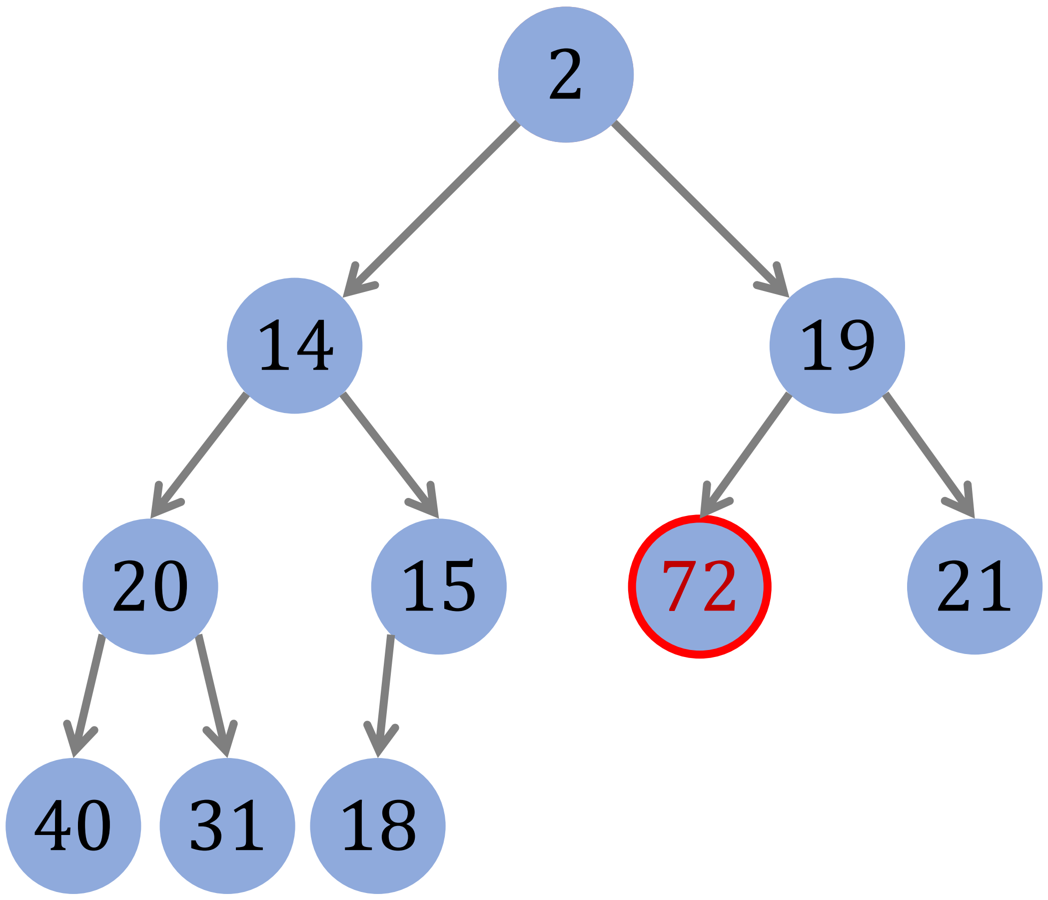
Tree 2:

9	50	14	76	66	22	43	92	88	68	69	24

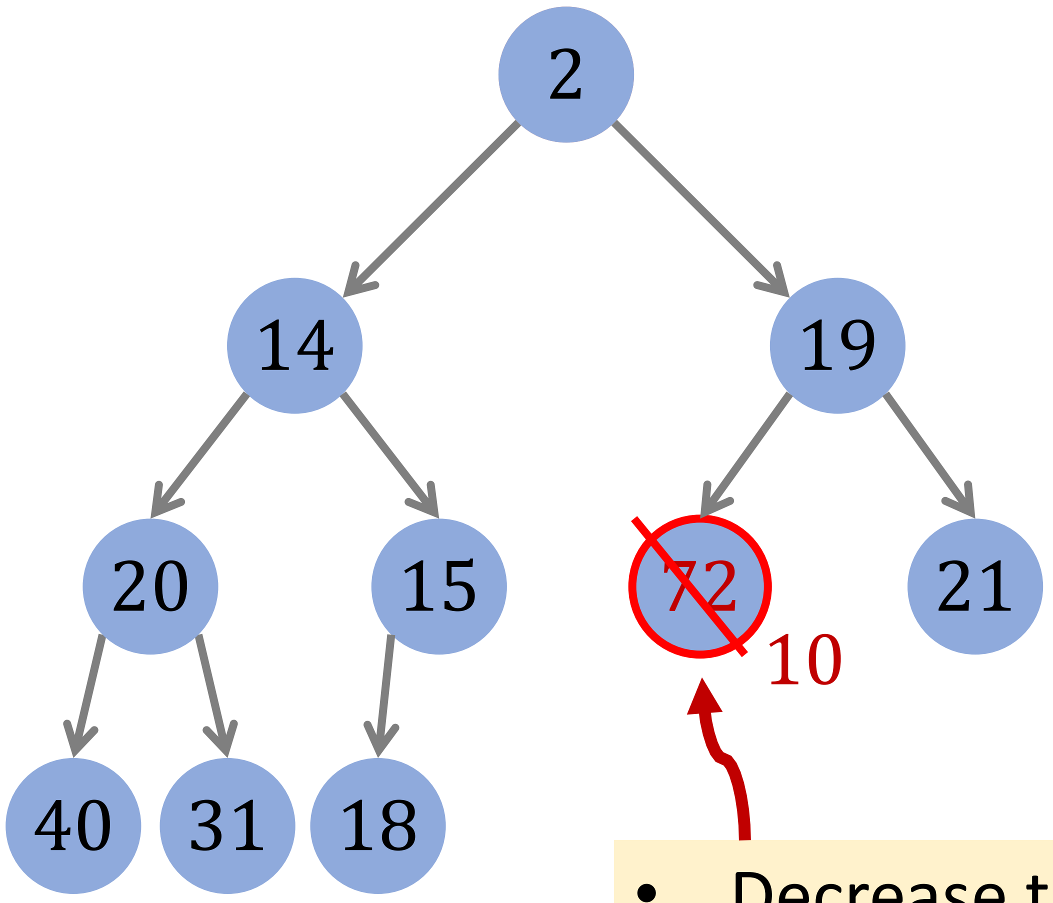
Tree 3:

7	19	26	36	22	98	42	43	55	23		

Q4: decreaseKey()



Q4: decreaseKey()



- Decrease this key from 72 to 10.
- How to maintain the heap's property?

Thank You!