[SWE2015-41] Introduction to Data Structures (자료구조개론)

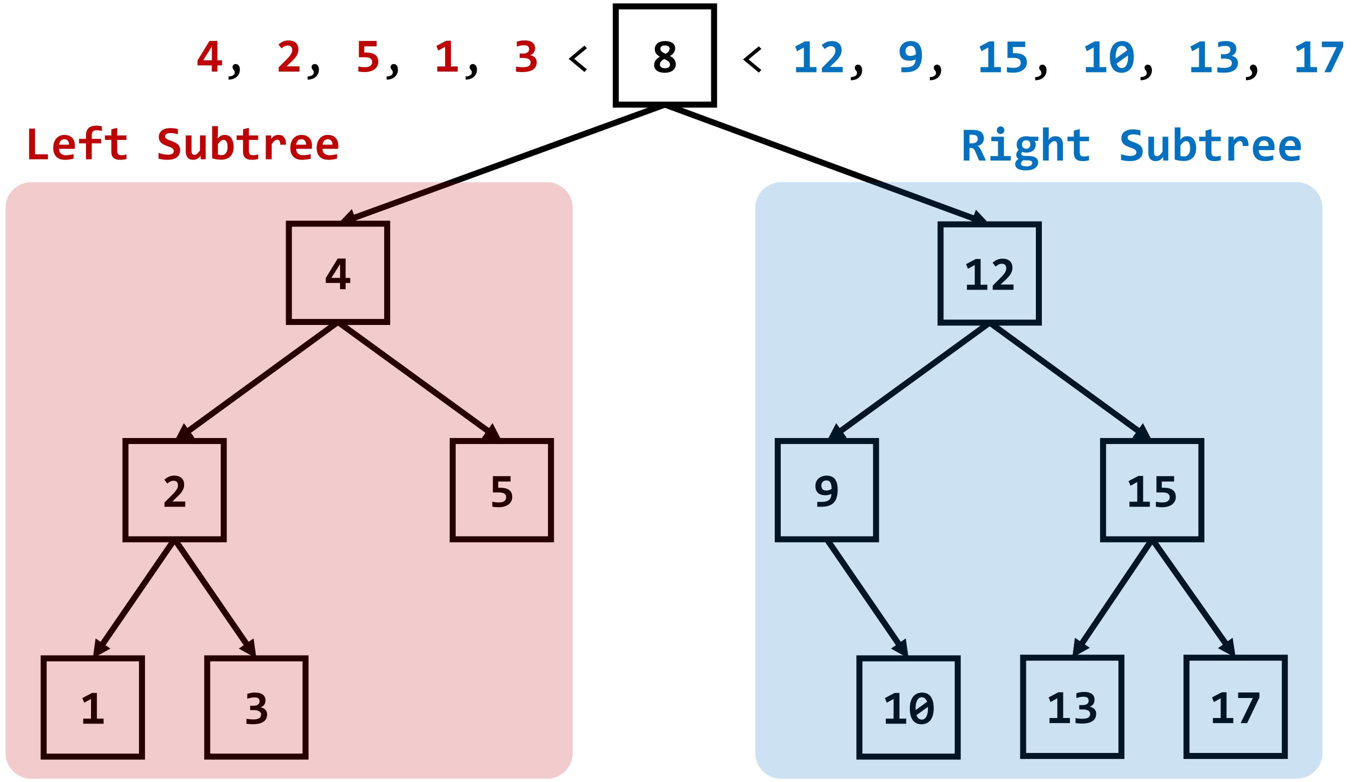
Heap

Department of Computer Science and Engineering

Instructor: Hankook Lee (이한국)

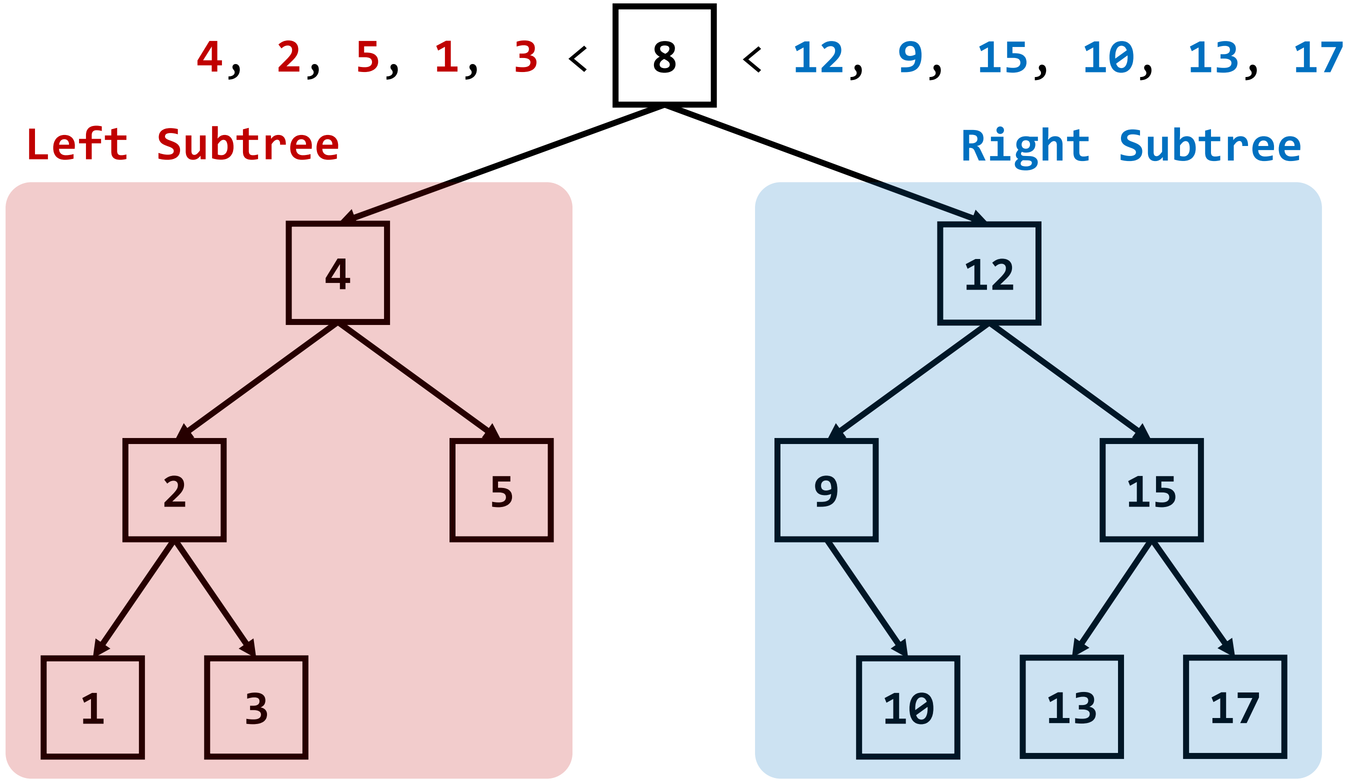
(Recap) Binary Search Trees (BSTs)

- **Binary Search Trees (BSTs)** are efficient for search, insertion, deletion, …
  - Due to the relationship between root, left subtree, and right subtree



(Recap) Binary Search Trees (BSTs)

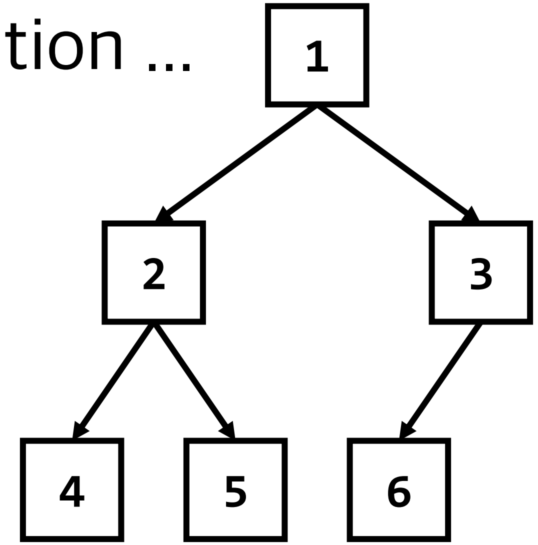
- **Binary Search Trees (BSTs)** are efficient for search, insertion, deletion, …
  - Due to the relationship between root, left subtree, and right subtree
  - But its implementation is complicated … (e.g., AVL & Red-Black trees)



(Recap) Binary Search Trees (BSTs)

- **Binary Search Trees (BSTs)** are efficient for search, insertion, deletion, …
  - Due to the relationship between root, left subtree, and right subtree
  - But its implementation is complicated … (e.g., AVL & Red-Black trees)

- There are **other options** when you don't need search operation …
  - Based on **complete binary trees** which supports …
    1. Easy representation (with array)
    2. Easy access between parent and children (with numbering)

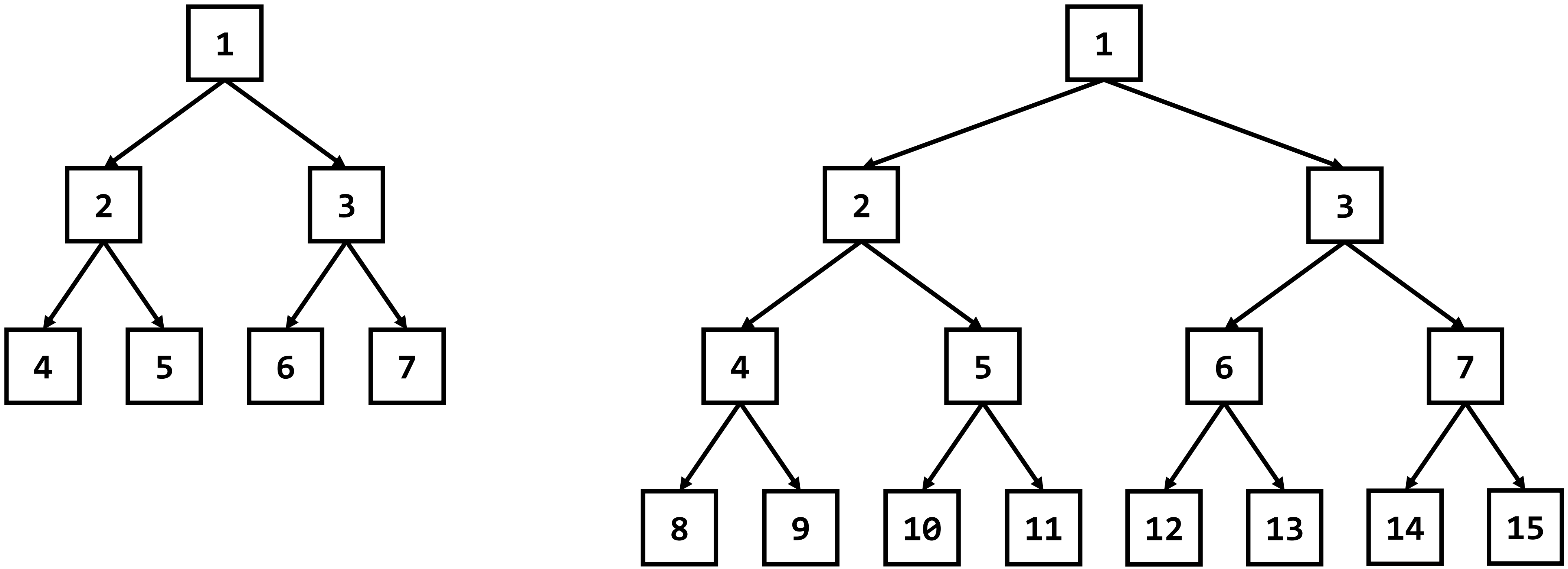


You can use …

- **Heap** if interested in only the minimum or maximum element
- **Segment Tree** if interested in statistics (e.g., sum, avg) of segments (intervals)

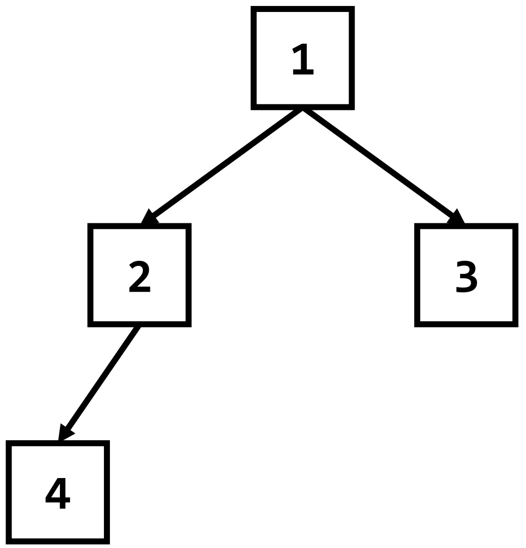
(Recap) Full & Complete Binary Trees

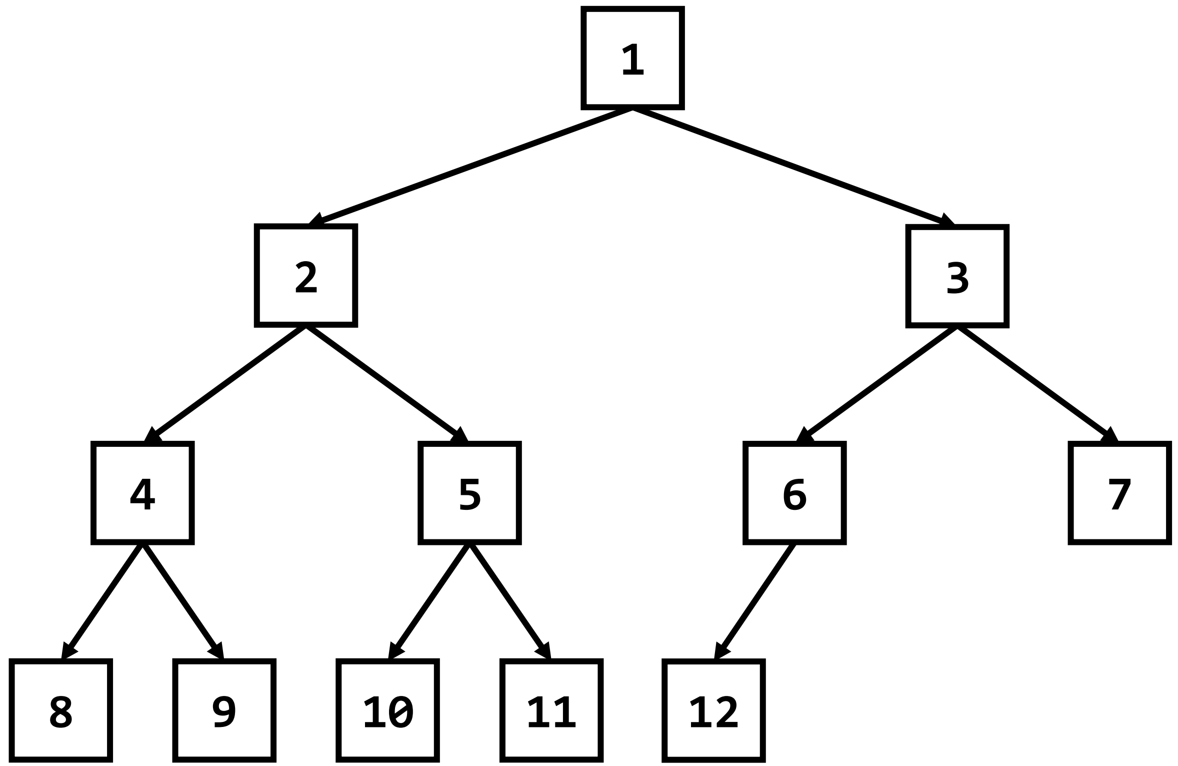
- **Full Binary Tree** is a BT of height H has $2^H - 1$ nodes
  - Node numbering from lower to higher levels, from left to right



(Recap) Full & Complete Binary Trees

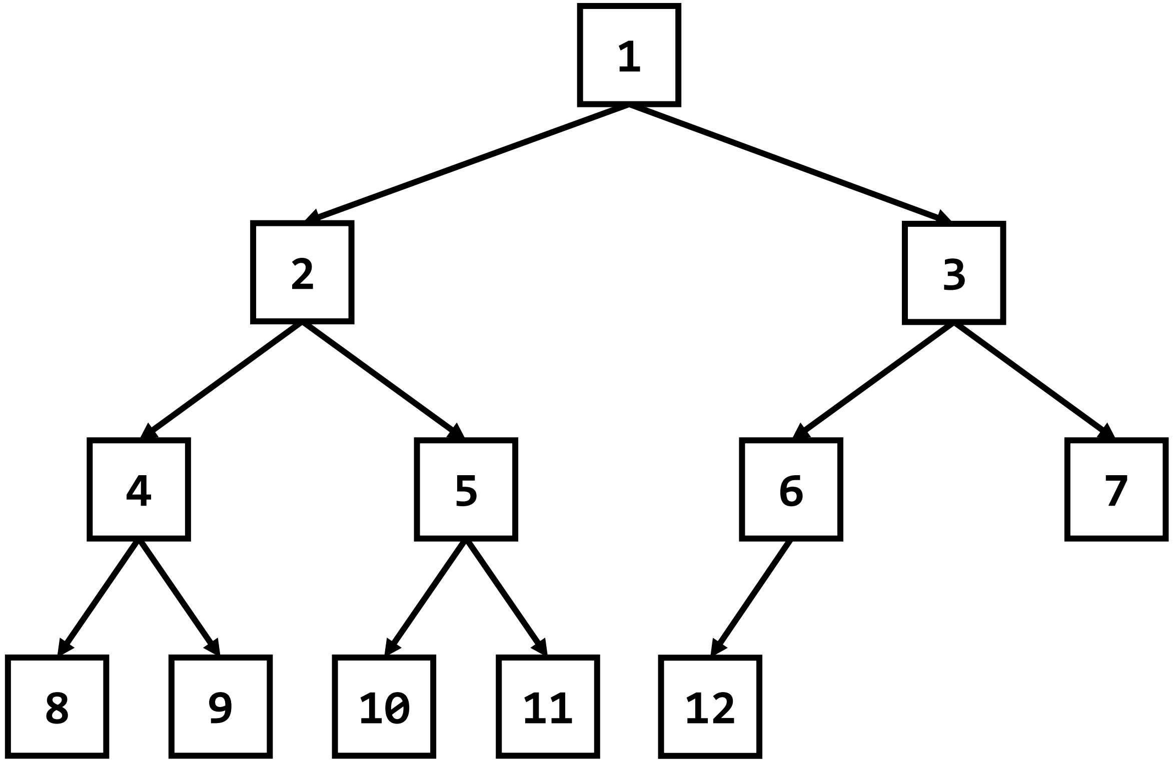
- **Complete Binary Tree** is a BT satisfying …
  - All nodes are sequentially filled from lower to higher levels, from left to right
  - The same node numbering to the full binary tree





(Recap) Complete Binary Tree

- The nodes in a complete binary tree are **sequentially filled**
  - There exists the **unique node numbering**
  - You can efficiently implement a complete BT using the array structure



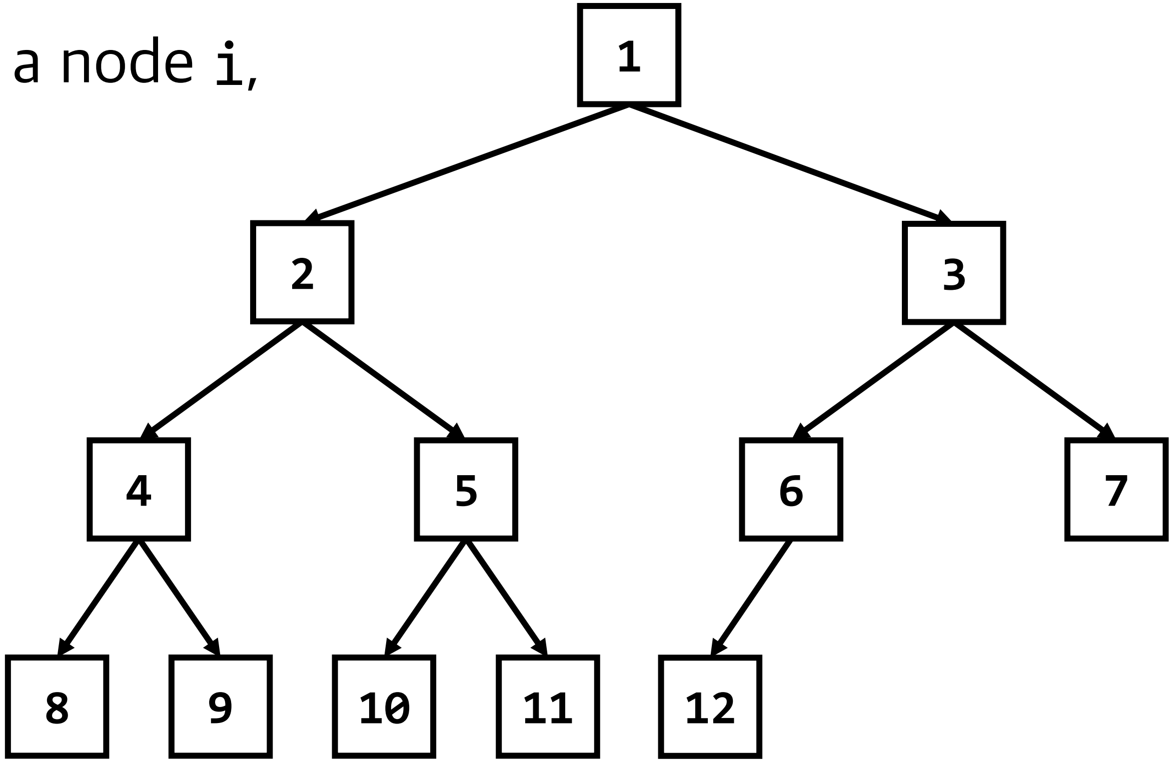
(Recap) Complete Binary Tree

- The nodes in a complete binary tree are **sequentially filled**
  - There exists the **unique node numbering**
  - You can efficiently implement a complete BT using the array structure

Interesting property of the numbering: for a node `i`,

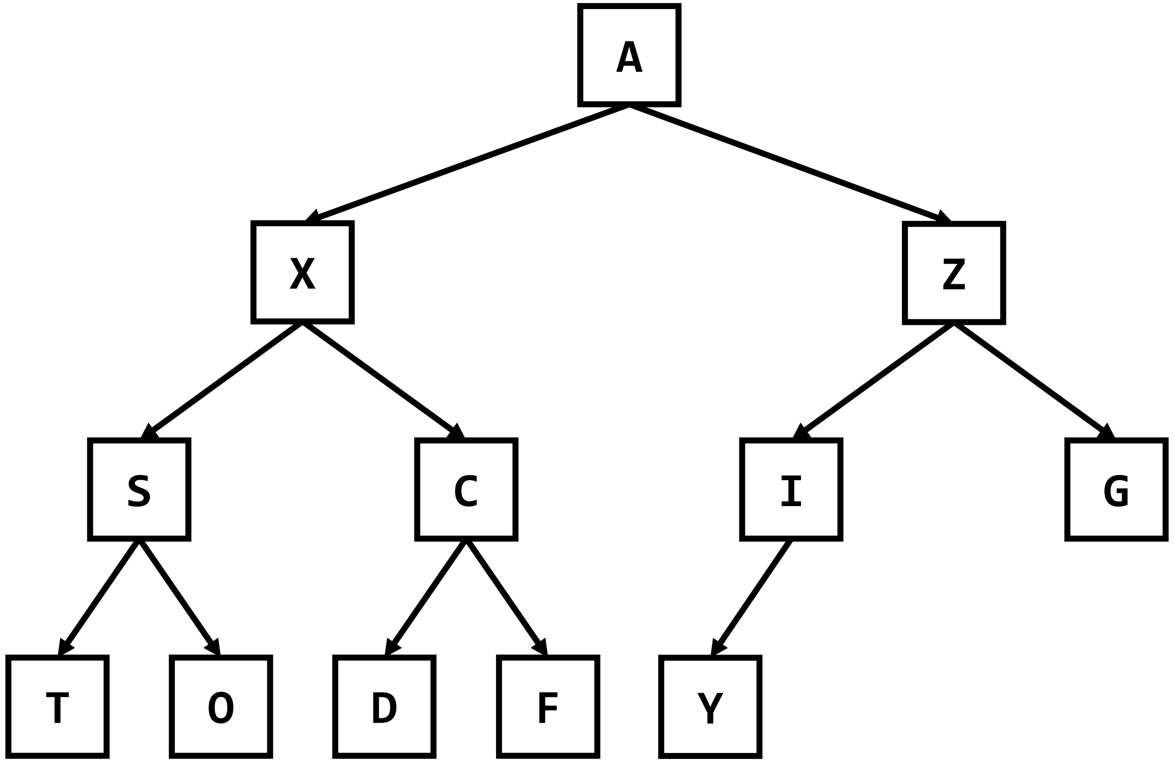
- its parent is `i/2`
- its left child is `i*2`
- its right child is `i*2+1`

- You can traverse nodes much easier



(Recap) Complete Binary Tree

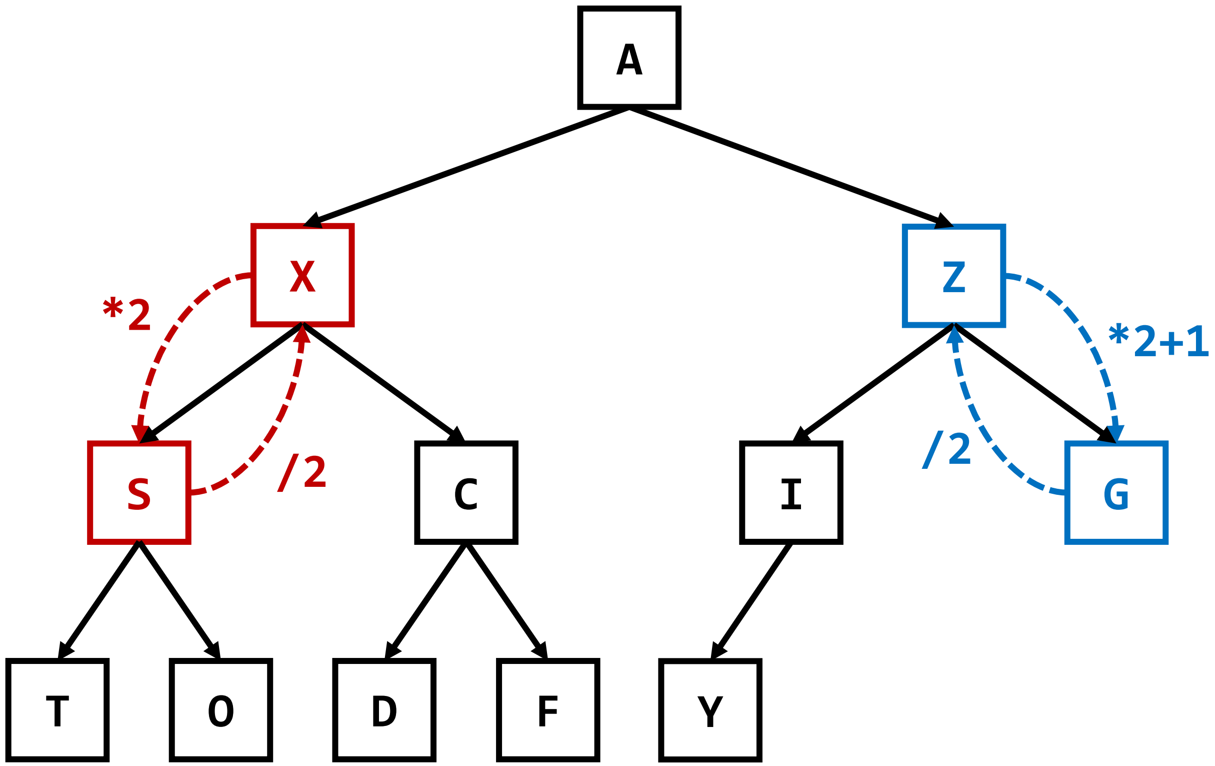
- A complete tree can be represented by an array structure



i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	A	X	Z	S	C	I	G	T	O	D	F	Y				

(Recap) Complete Binary Tree

- A complete tree can be represented by an array structure



i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
	A	X	Z	S	C	I	G	T	O	D	F	Y				

What is Heap?

- Heap is a complete binary tree satisfying …
  - Each node has its own **priority** (like key in BSTs)
  - Any node has a higher priority than its children:

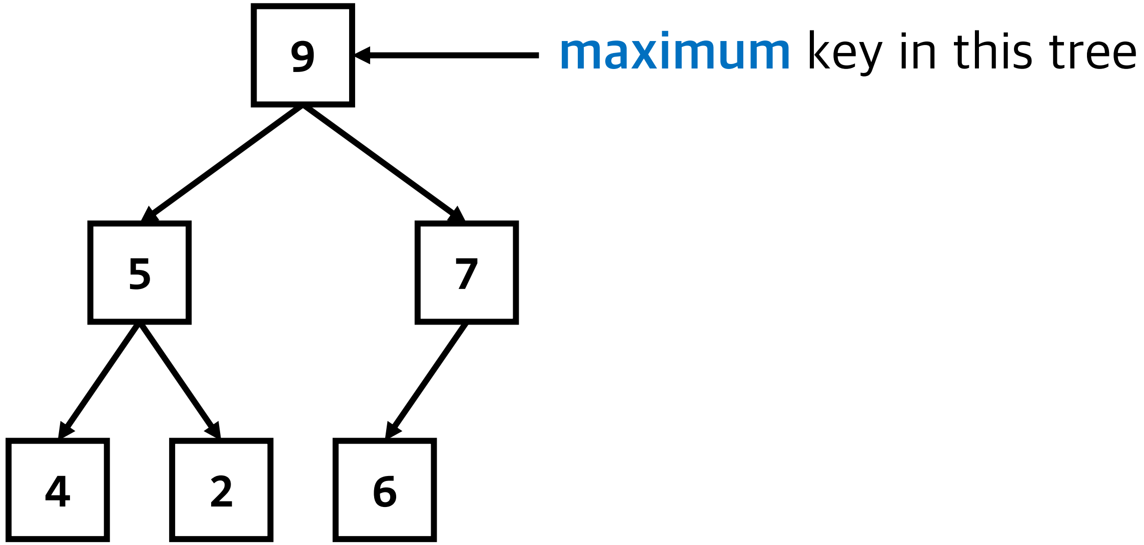
```
priority(parent) >= priority(child)
```

What is Heap?

- Heap is a complete binary tree satisfying …
  - Each node has its own **priority** (like key in BSTs)
  - Any node has a higher priority than its children:

```
priority(parent) >= priority(child)
```

  - If a larger key means a higher priority … (**MAX** heap)

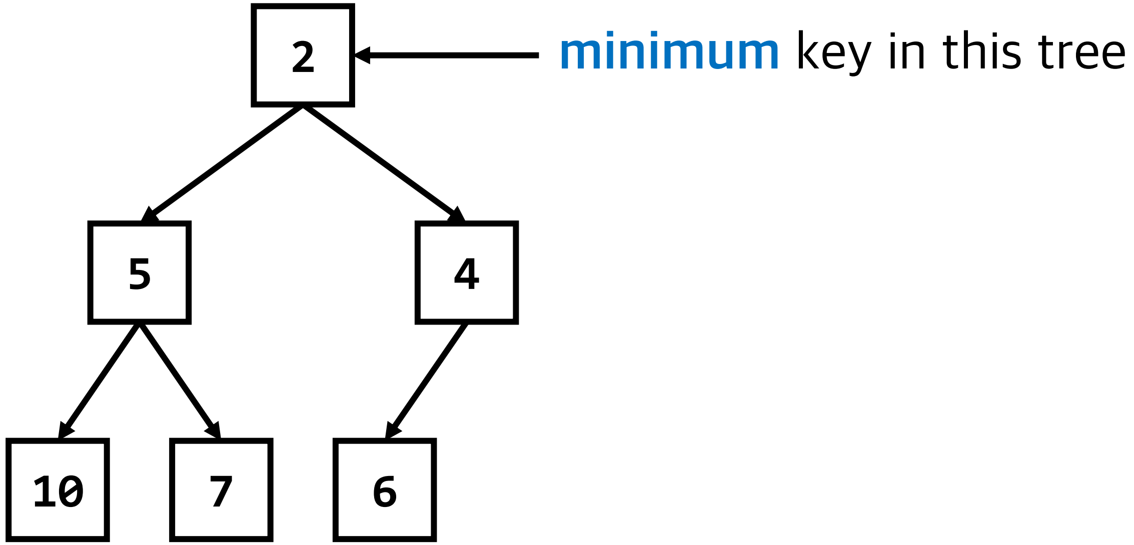


What is Heap?

- Heap is a complete binary tree satisfying …
  - Each node has its own **priority** (like key in BSTs)
  - Any node has a higher priority than its children:

```
priority(parent) >= priority(child)
```

  - If a smaller key means a higher priority … (**MIN** heap)



What is Heap?

- Heap is a complete binary tree satisfying …
  - Each node has its own **priority** (like key in BSTs)
  - Any node has a higher priority than its children:

```
priority(parent) >= priority(child)
```

- **Properties**
  - The height is $O(\log N)$ where N is the number of nodes
  - The root node has the highest priority
    - You can find the most important node in $O(1)$

What is Heap?

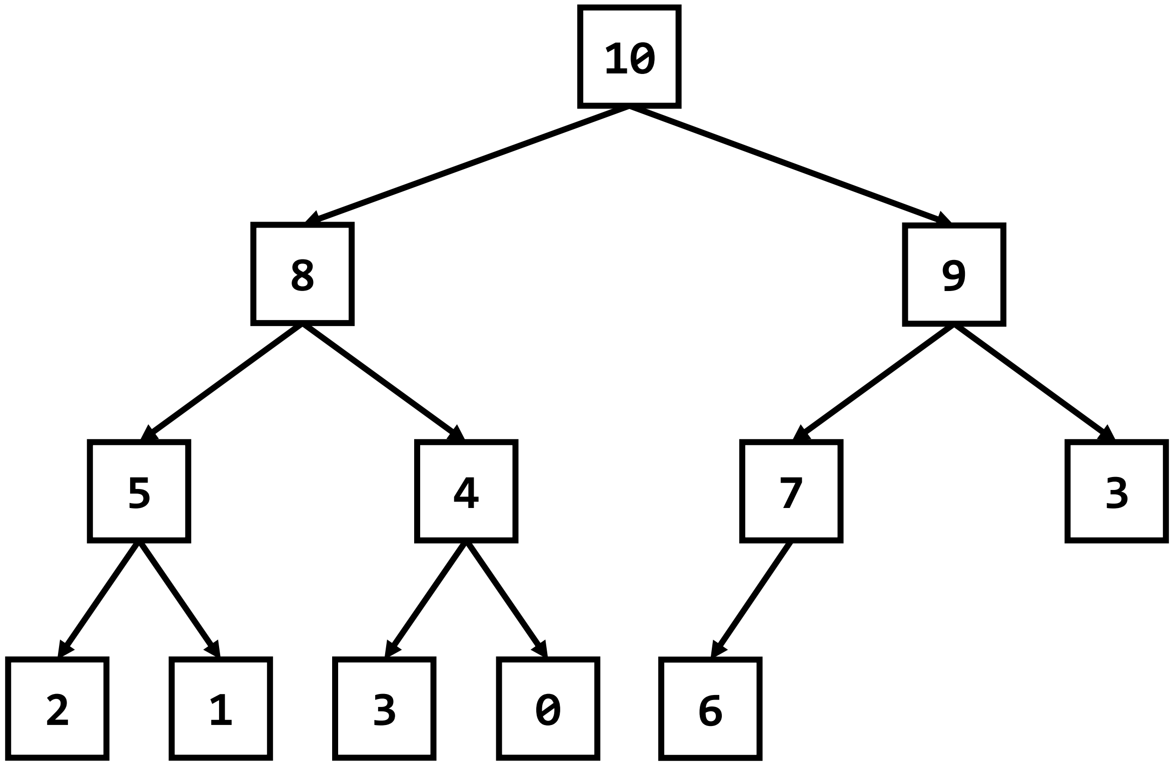
- Heap is a complete binary tree satisfying …
  - Each node has its own **priority** (like key in BSTs)
  - Any node has a higher priority than its children:

```
priority(parent) >= priority(child)
```

- **Properties**
  - The height is $O(\log N)$ where N is the number of nodes
  - The root node has the highest priority
    - You can find the most important node in $O(1)$
  - Insertion of a new node has $O(\log N)$ time complexity
  - Deletion of the root node has $O(\log N)$ time complexity
  - Note. Heap does not support **efficient** search operation

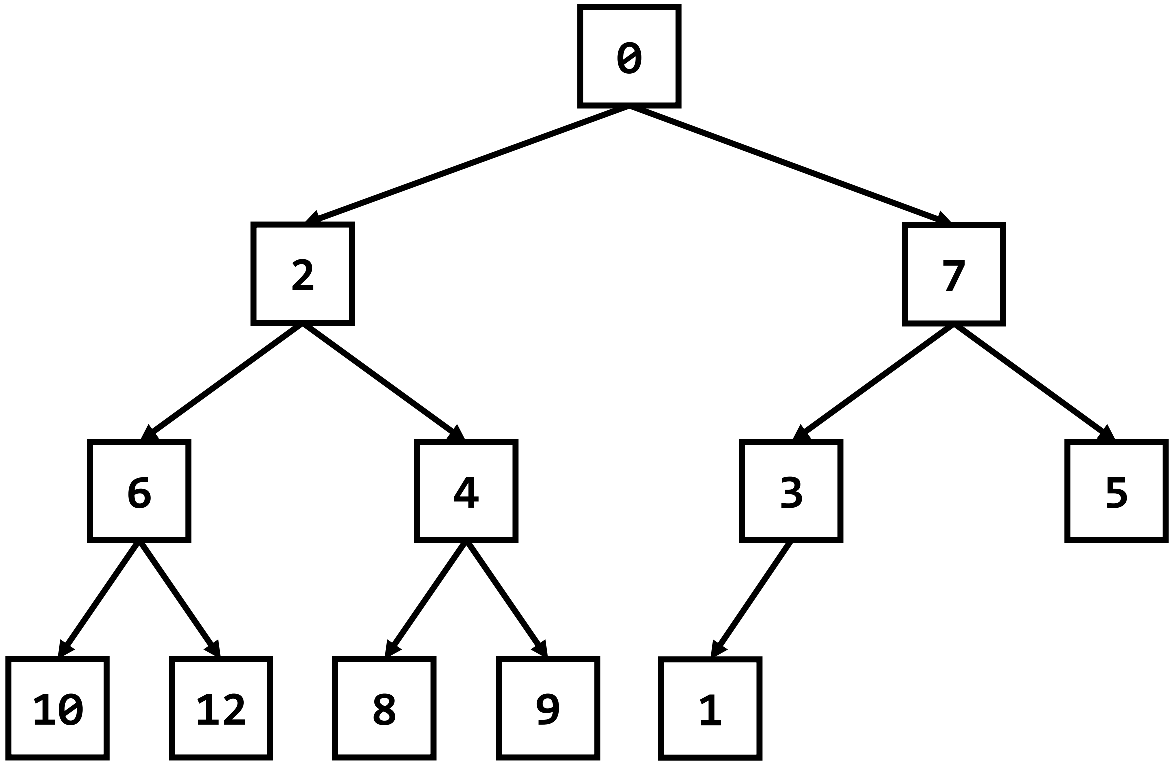
Examples

- Max heap



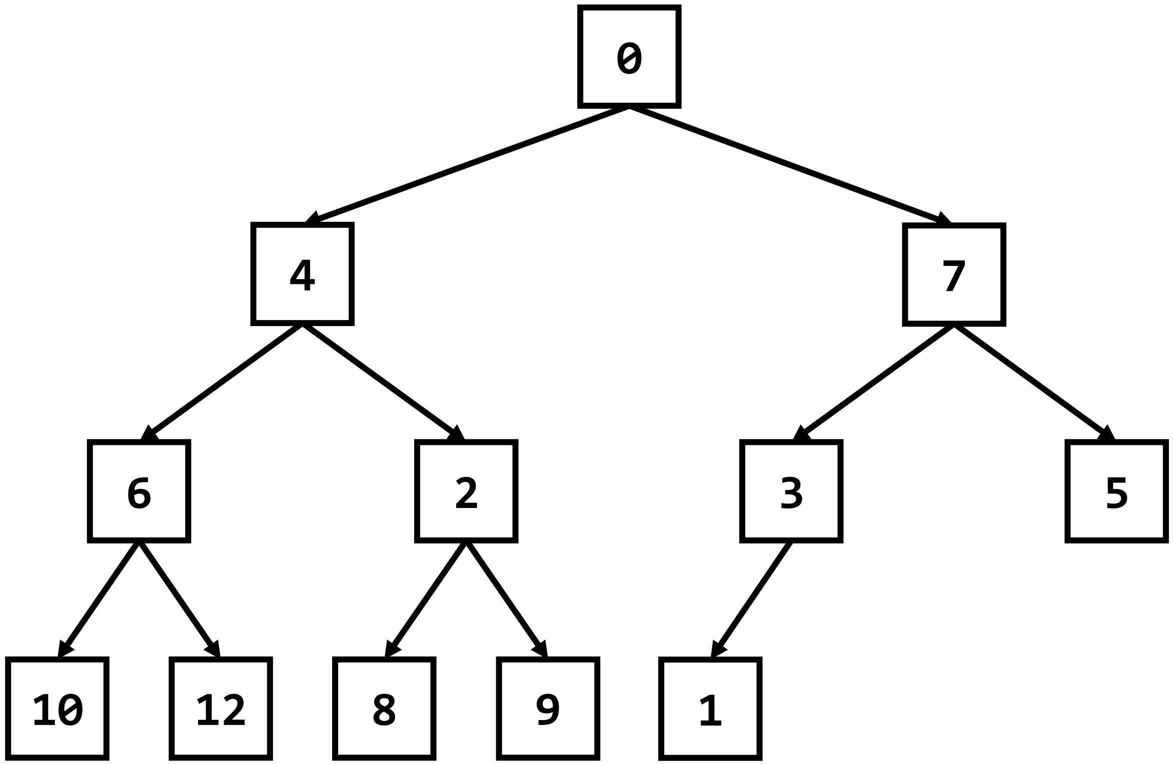
Examples

- Min heap



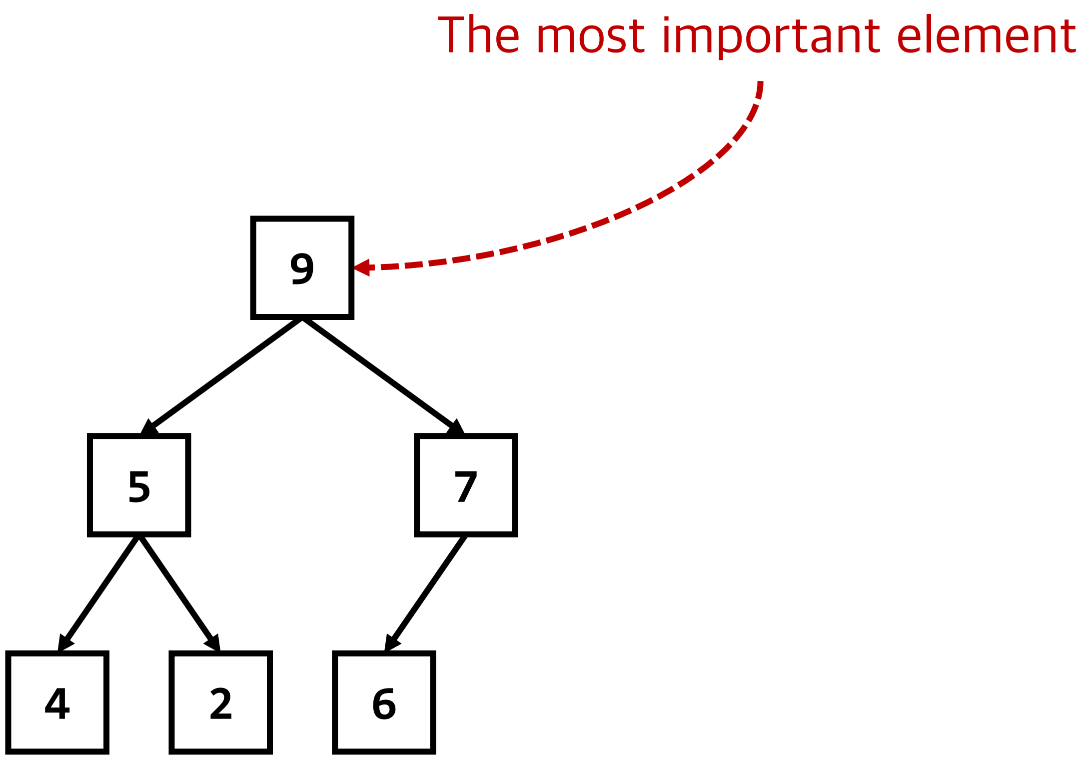
Examples

- NOT heap



Heap Operations

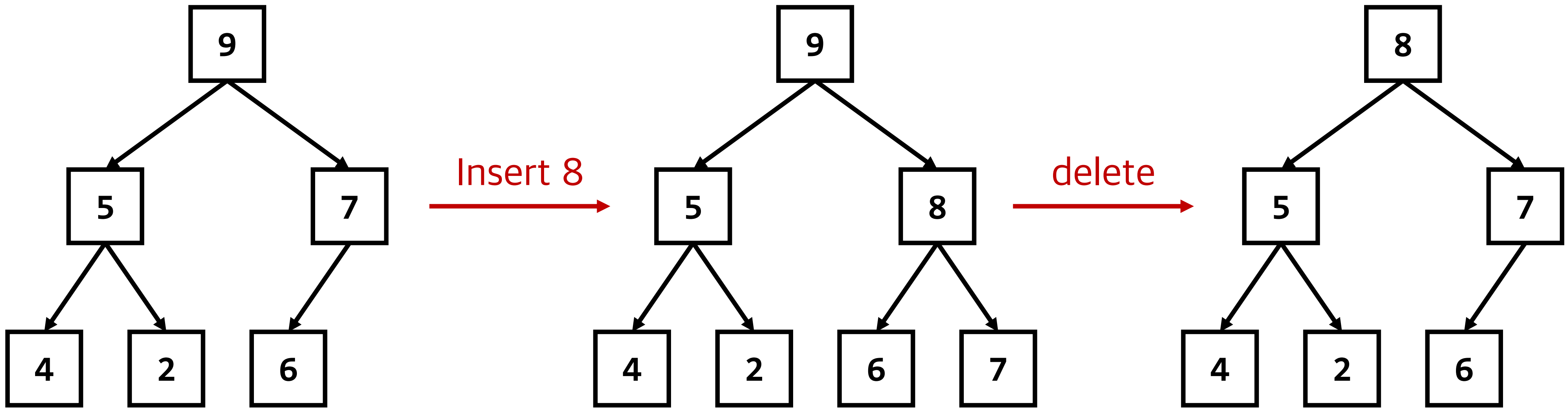
- `getFirst` - find the first (i.e., the most important) element
  - The element is always the root node
  - This is similar to the `peek()` operation in queue and stack



Heap Operations

- `getFirst` - find the first (i.e., the most important) element
  - The element is always the root node
  - This is similar to the `peek()` operation in queue and stack

- `insert` - insert an element without violating the priority condition
- `delete` - delete the root node without violating the priority condition



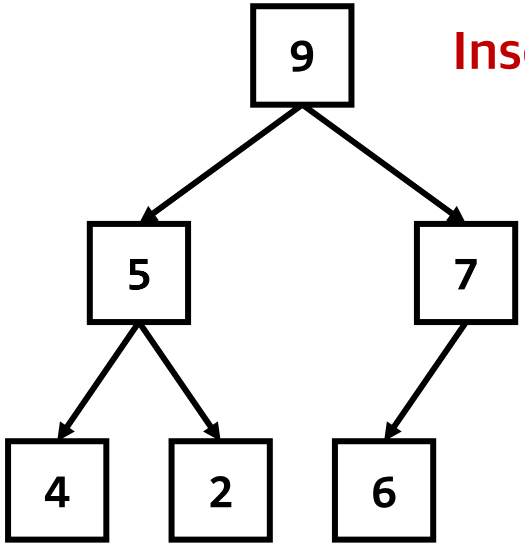
Heap Operations

- Recommend using the **array-based** representation for the heap structure
  - In this lecture, we'll focus on max heap. Min heap implementation is very similar

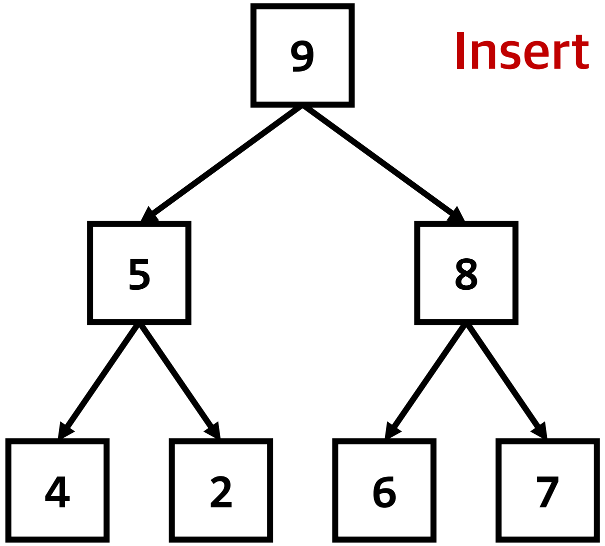
```c
typedef struct _MaxHeap {
    int items[MAX_SIZE+1];
    int size;
} MaxHeap;

int getFirst(MaxHeap *heap);
void insert(MaxHeap *heap, int item);
void delete(MaxHeap *heap);

int getFirst(MaxHeap *heap) {
    return heap->items[1]; // node numbering starts from 1
}
```

Heap - Insertion

- How to insert a new node into the heap?



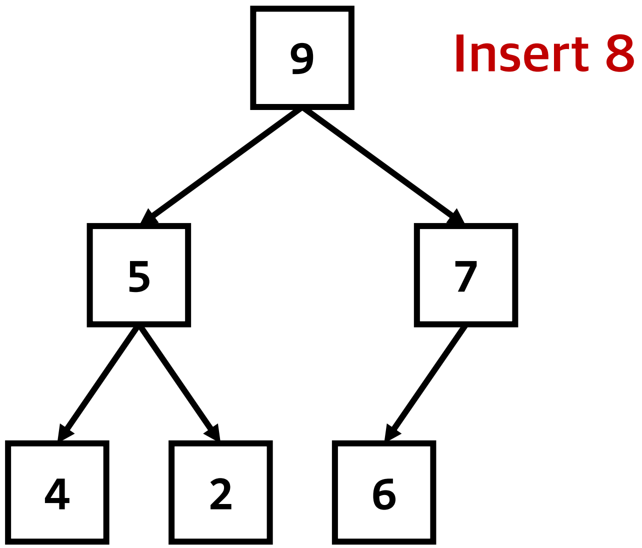
Insert 8

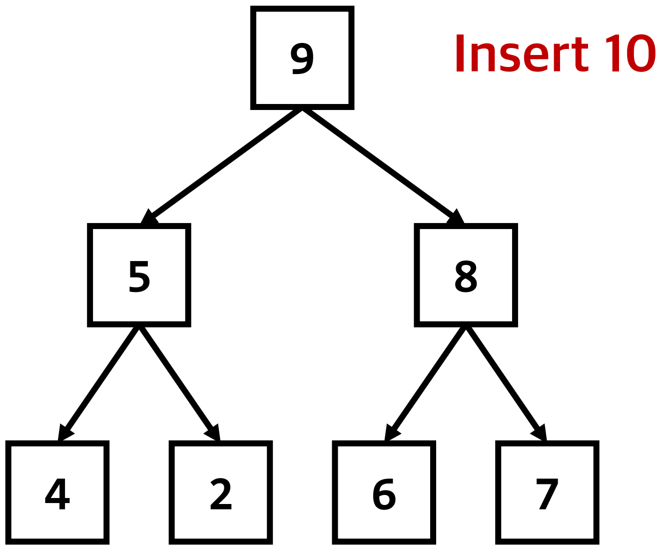


Insert 10

Heap - Insertion

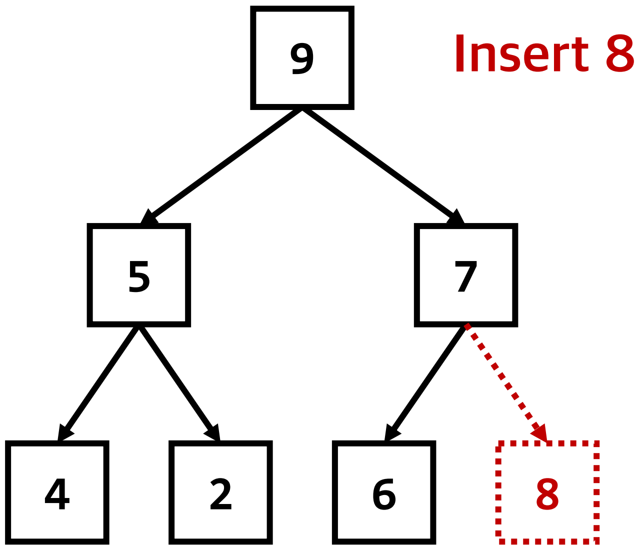
- How to insert a new node into the heap?
  (Step 1) Insert the node at **the last position** (i.e., bottom-rightmost)

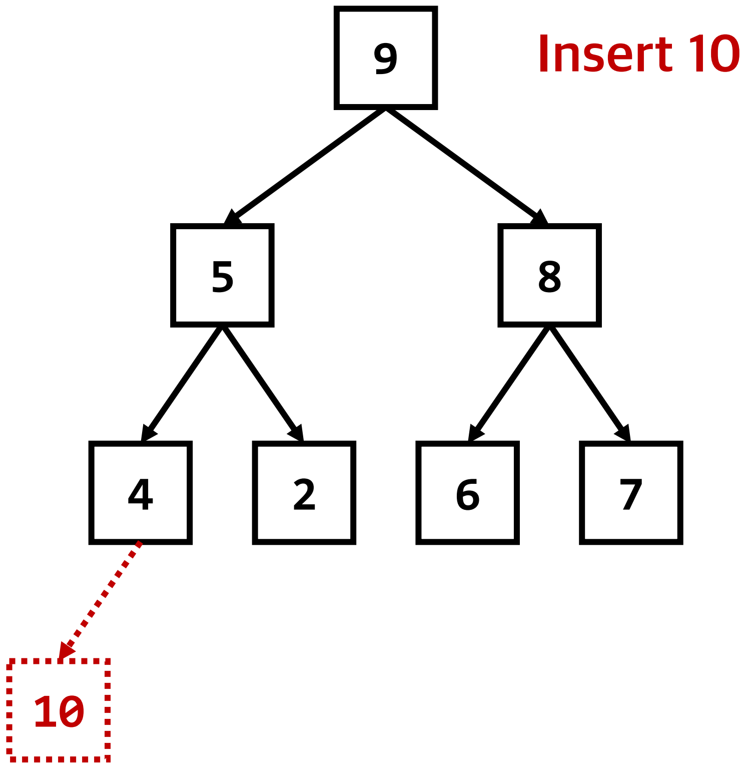




Heap - Insertion

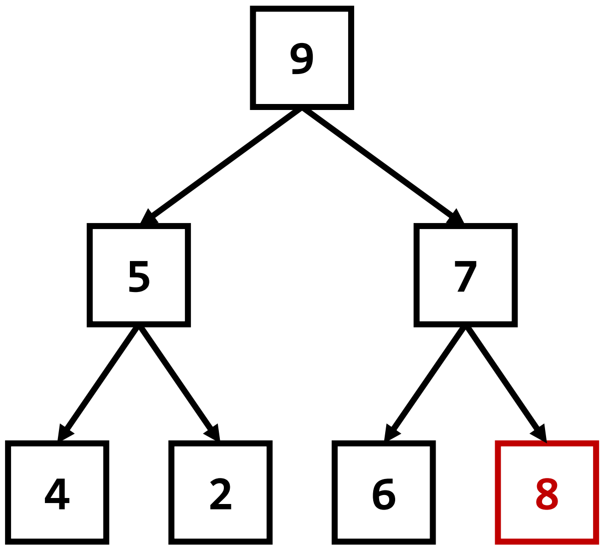
- How to insert a new node into the heap?
  - **(Step 1)** Insert the node at **the last position** (i.e., bottom-rightmost)





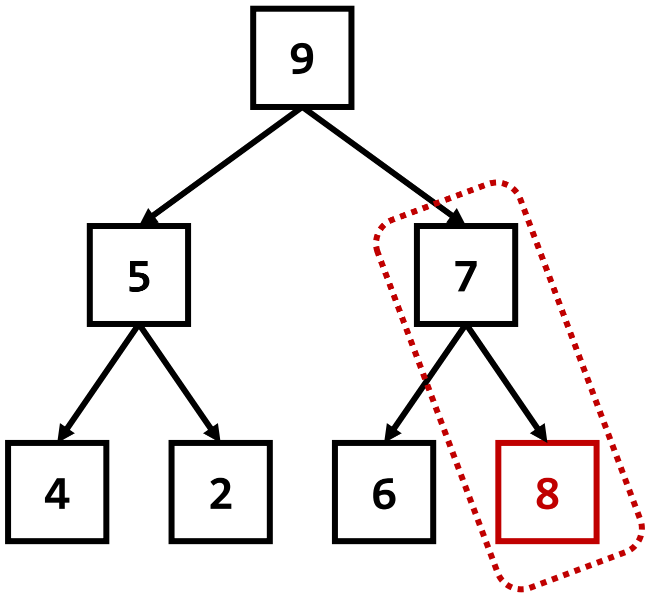
Heap - Insertion

- How to insert a new node into the heap?
  - **(Step 2)** If the **new node** and its parent violate the priority condition, swap them



Heap - Insertion

- How to insert a new node into the heap?
  - `(Step 2)` If the **new node** and its parent violate the priority condition, swap them



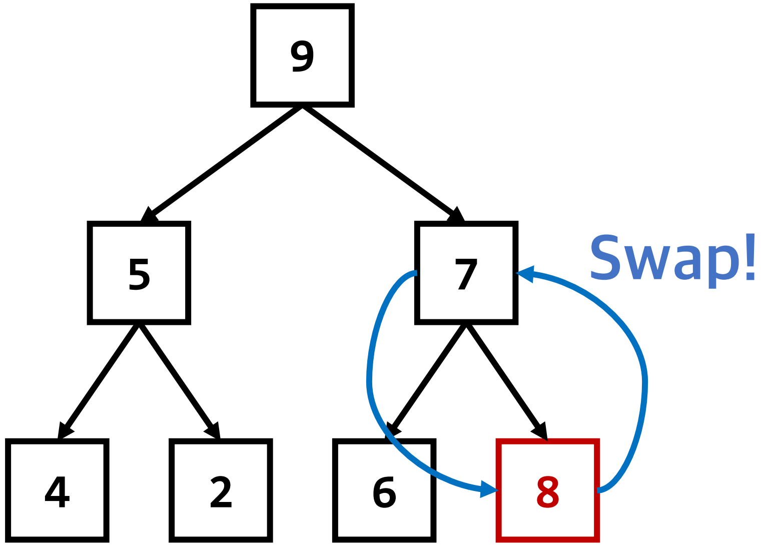
Priority condition is violated

`priority(parent) < priority(node)`

Heap - Insertion

- How to insert a new node into the heap?

  (Step 2) If the **new node** and its parent violate the priority condition, swap them

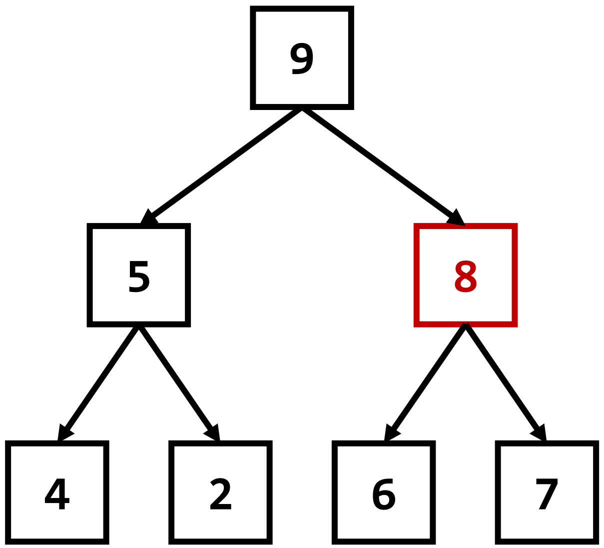


Priority condition is violated

`priority(parent) < priority(node)`

Heap - Insertion

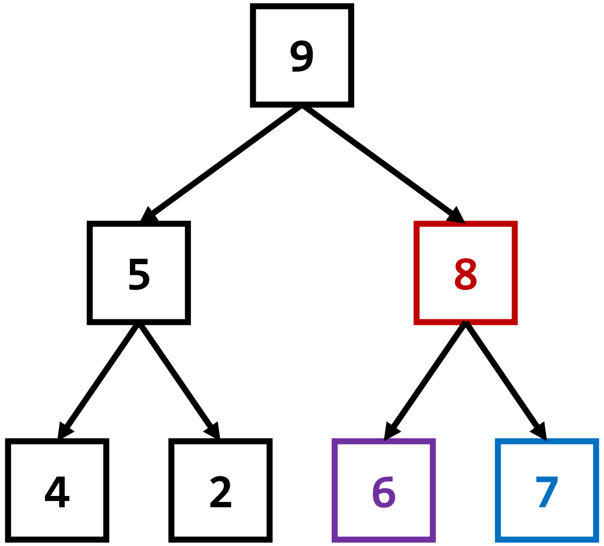
- How to insert a new node into the heap?
  `(Step 2)` If the **new node** and its parent violate the priority condition, swap them



Heap - Insertion

- How to insert a new node into the heap?

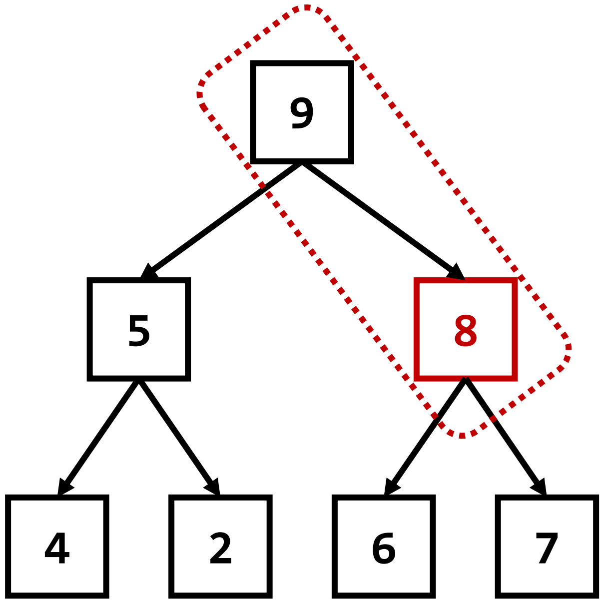
    (Step 2) If the **new node** and its parent violate the priority condition, swap them



You don't need to compare the **new node** with **another child** (in this case, 6) because the **original parent** (in this case, 7) has a higher priority than **the child**

Heap - Insertion

- How to insert a new node into the heap?
  - **(Step 2)** If the **new node** and its parent violate the priority condition, swap them
    - Repeat this step until not violated

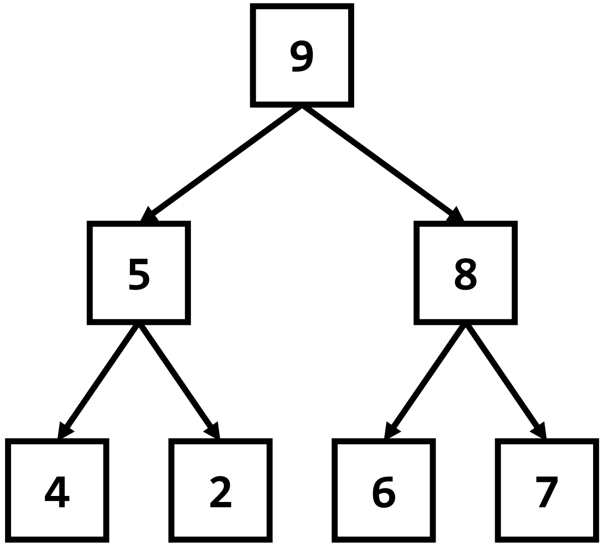


Priority condition is not violated

`priority(parent) > priority(node)`

Heap - Insertion

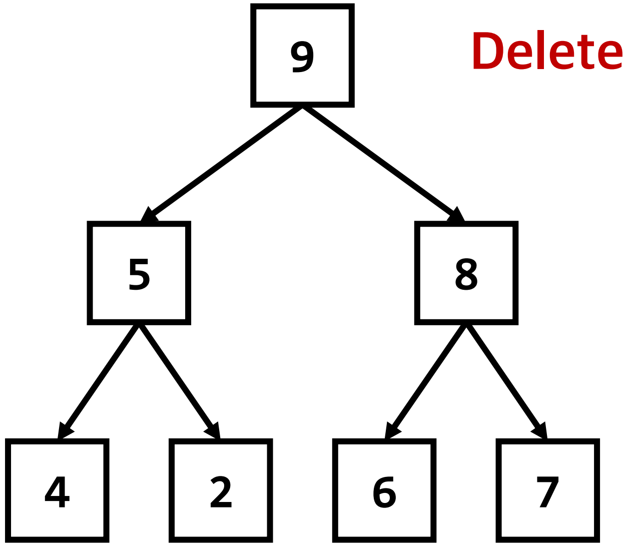
- How to insert a new node into the heap?
  (Step 2) If the **new node** and its parent violate the priority condition, swap them
    - Repeat this step until not violated



Done

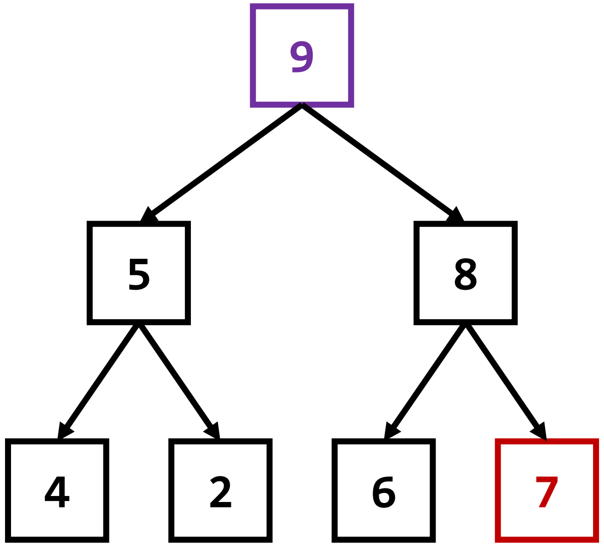
Heap - Deletion

- How to delete the root node?



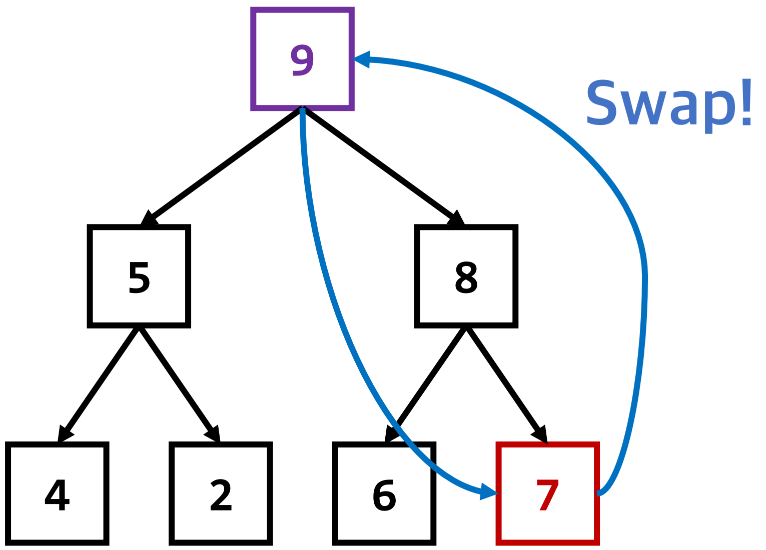
Heap - Deletion

- How to delete the root node?
  - **(Step 1)** Swap the **root node** and **the last node** & delete the **root node**



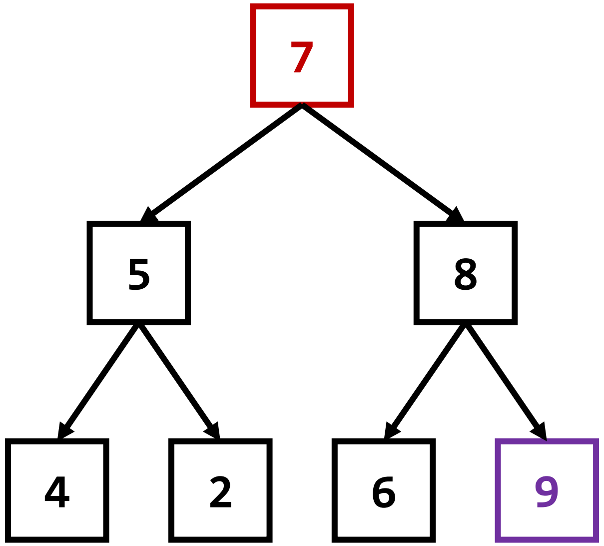
Heap - Deletion

- How to delete the root node?
    - **(Step 1)** Swap the **root node** and **the last node** & delete the **root node**



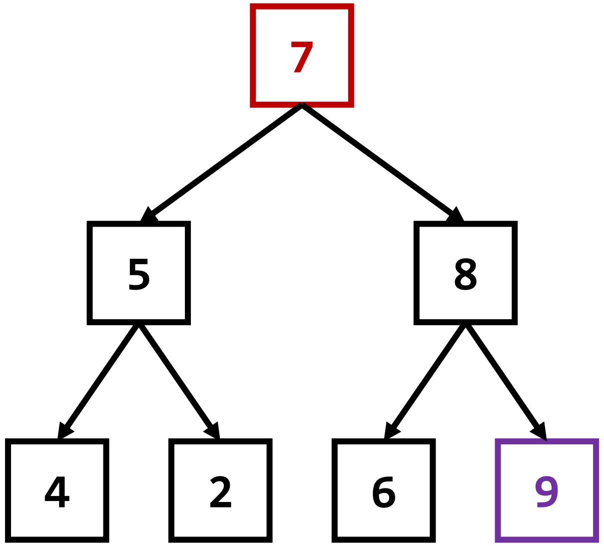
Heap - Deletion

- How to delete the root node?
  - **(Step 1)** Swap the **root node** and **the last node** & delete the **root node**



Heap - Deletion

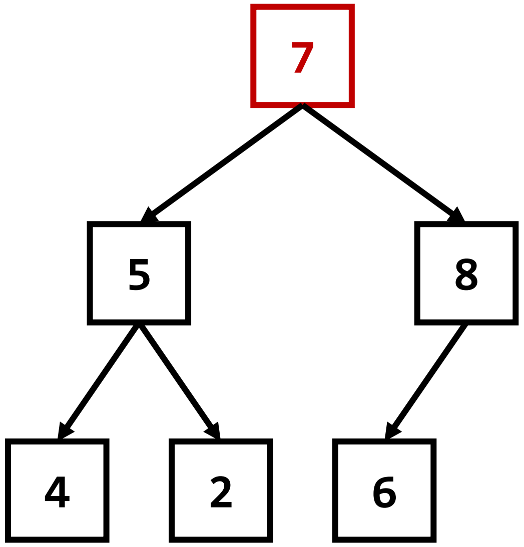
- How to delete the root node?
  - **(Step 1)** Swap the **root node** and **the last node** & delete the **root node**



Delete this

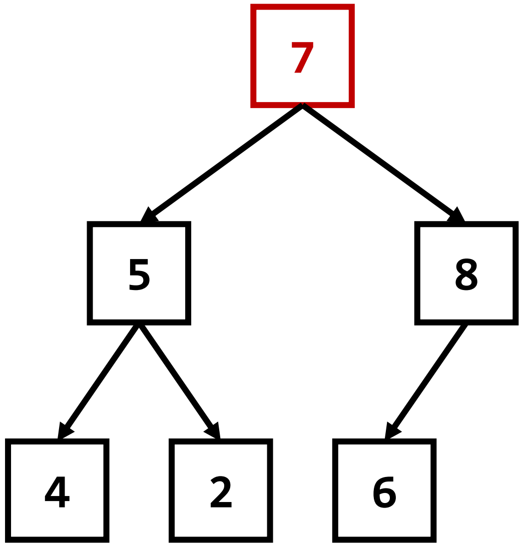
Heap - Deletion

- How to delete the root node?
  - **(Step 1)** Swap the **root node** and **the last node** & delete the **root node**



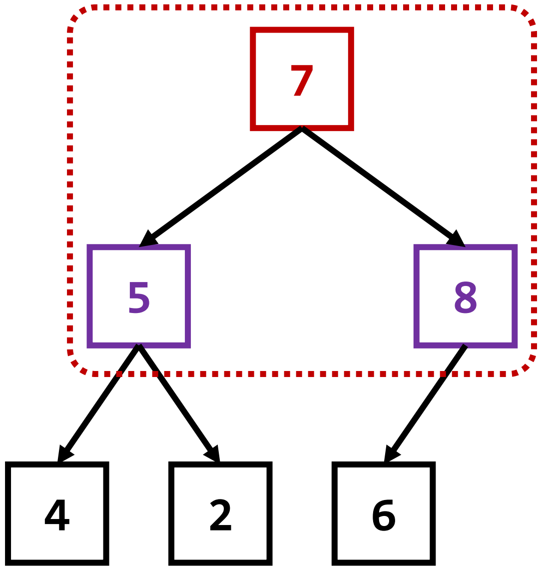
Heap - Deletion

- How to delete the root node?
  - **(Step 2)** If the **last node** and **its children** violate the priority condition, swap them



Heap - Deletion

- How to delete the root node?
  - **(Step 2)** If the **last node** and **its children** violate the priority condition, swap them



Priority condition is violated

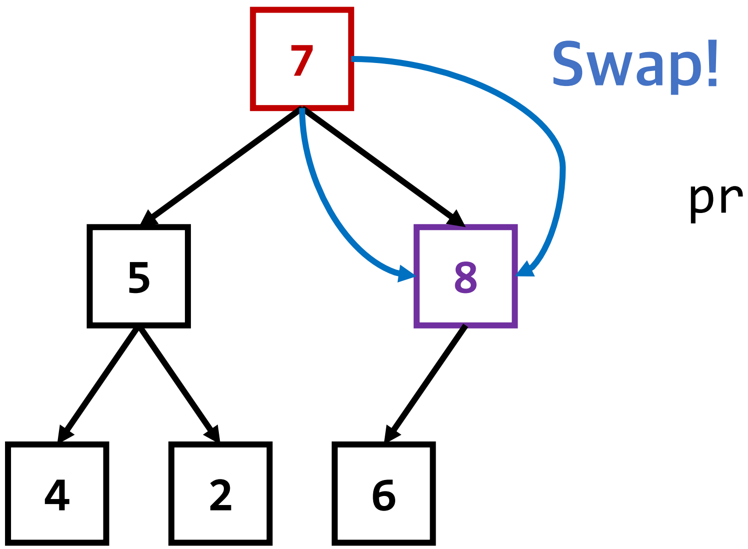
`priority(parent) < max(priority(child))`

Heap - Deletion

- How to delete the root node?

  (Step 2) If the **last node** and **its children** violate the priority condition, swap them

  - Swap the **last node** and **the child of the highest priority**



Priority condition is violated

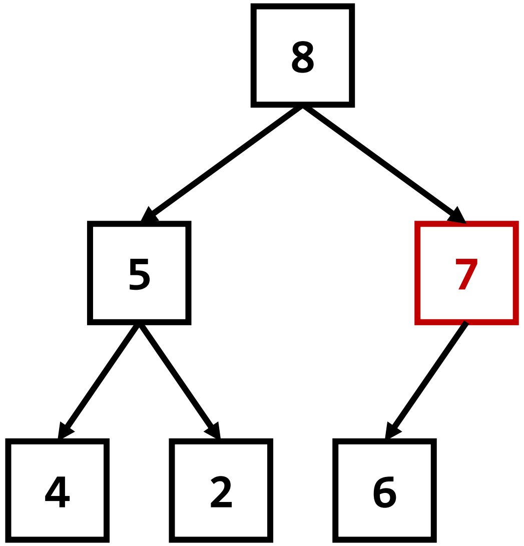
`priority(parent) < max(priority(child))`

Heap - Deletion

- How to delete the root node?

  (Step 2) If the last node and its children violate the priority condition, swap them

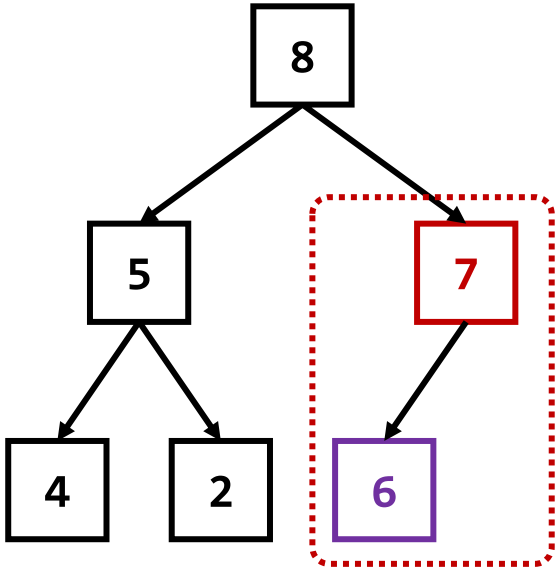
  - Swap the last node and the child of the highest priority
  - Repeat this step until not violated



Heap - Deletion

- How to delete the root node?

  (Step 2) If the **last node** and **its children** violate the priority condition, swap them
  - Swap the **last node** and **the child of the highest priority**
  - Repeat this step until not violated

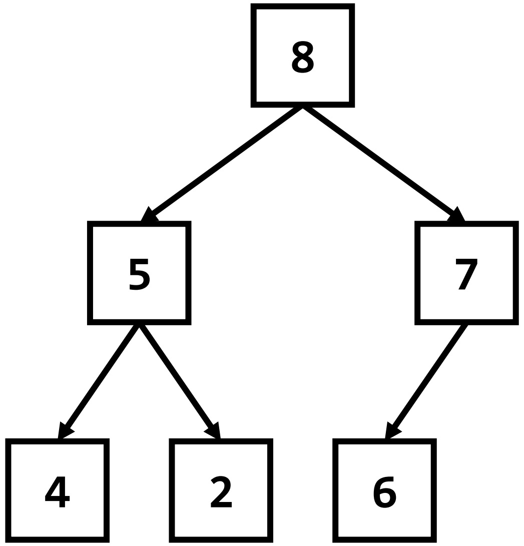


Priority condition is not violated

`priority(parent) > max(priority(child))`

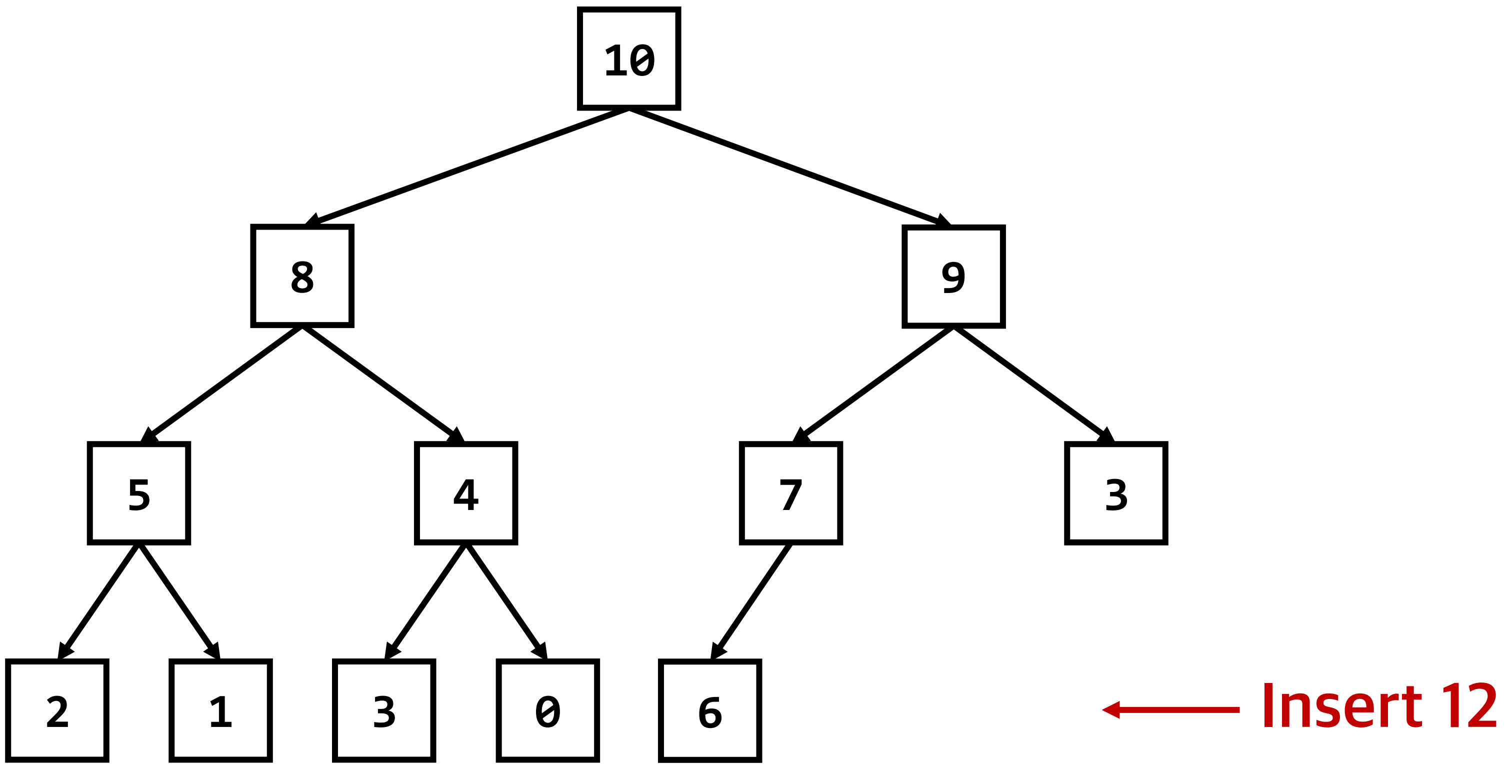
Heap - Deletion

- How to delete the root node?
  - **(Step 2)** If the **last node** and **its children** violate the priority condition, swap them
    - Swap the **last node** and **the child of the highest priority**
    - Repeat this step until not violated

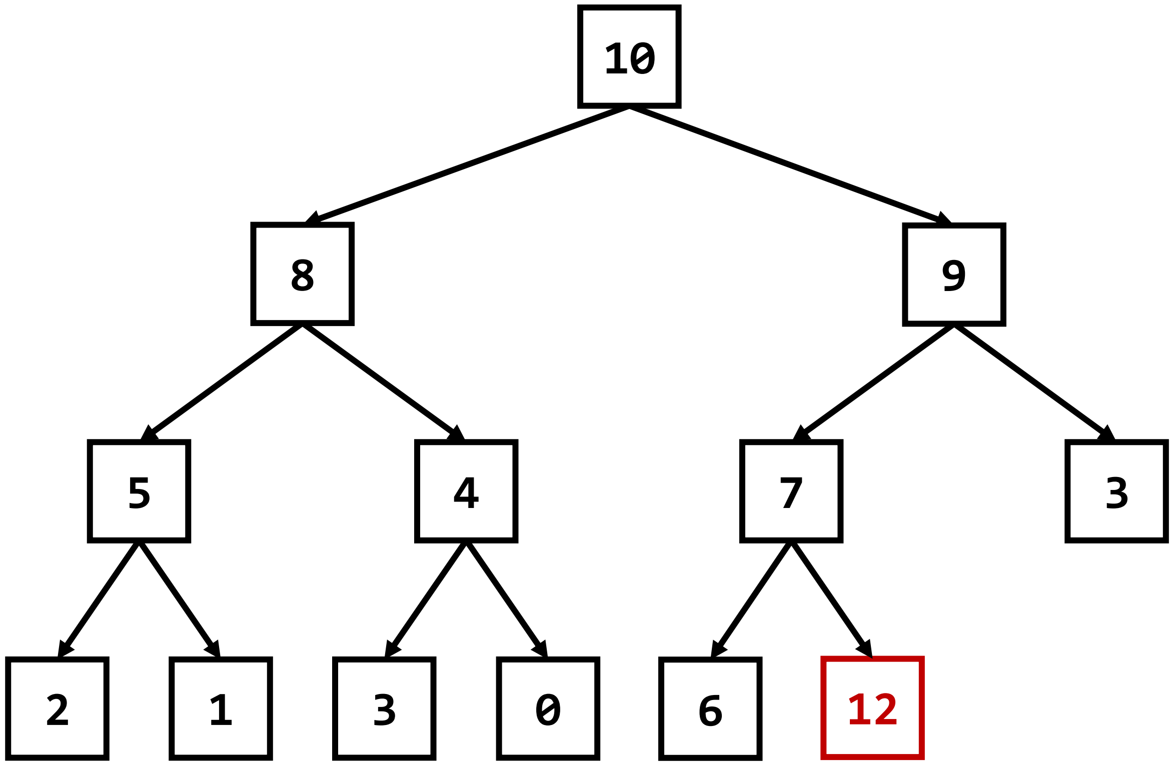


Done

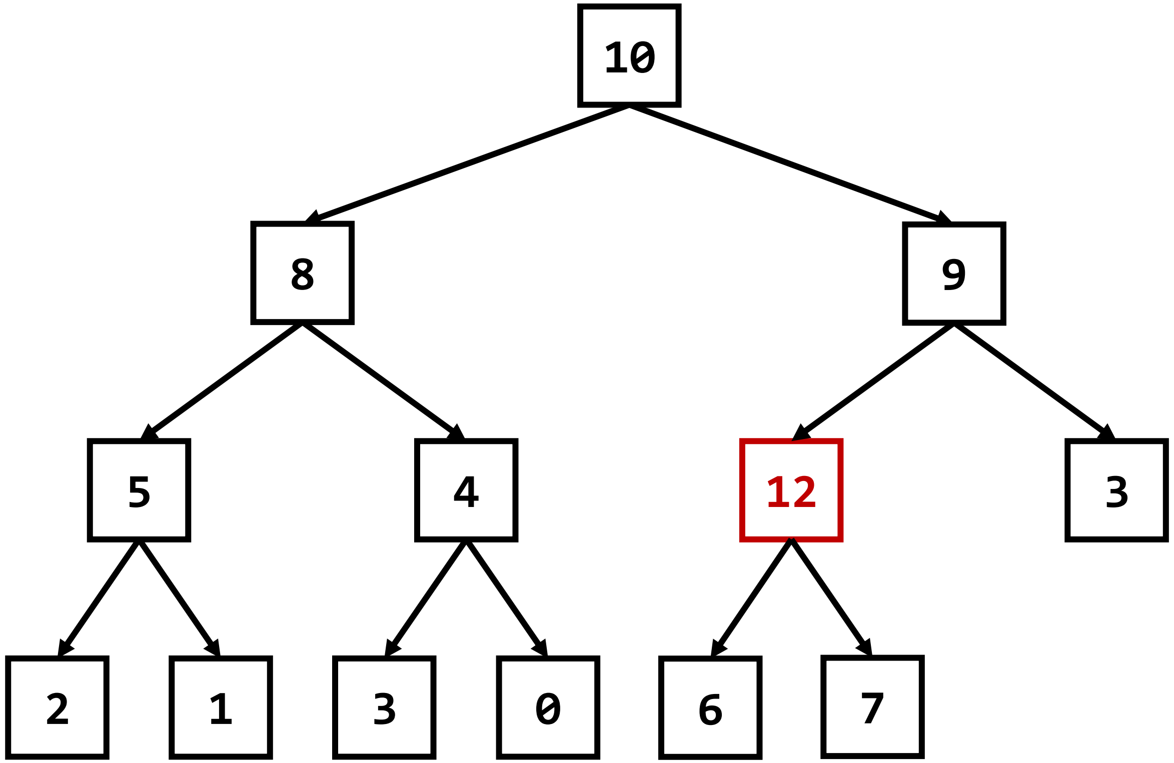
Examples



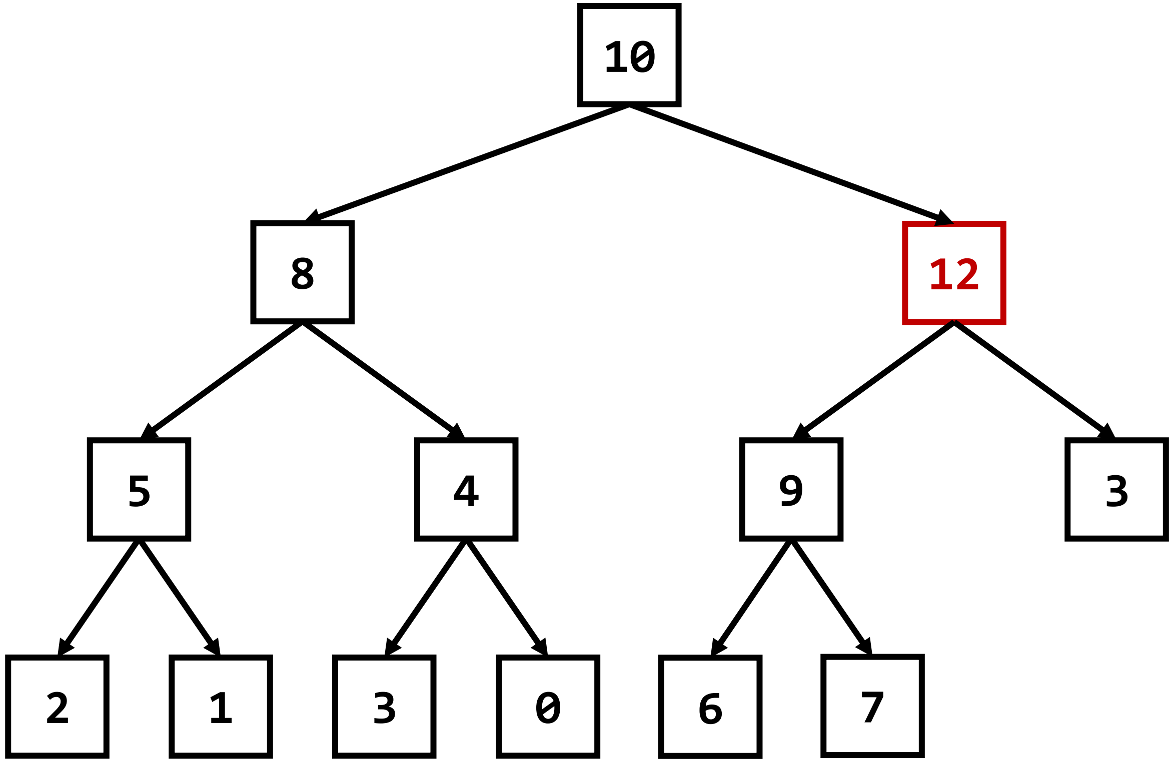
Examples



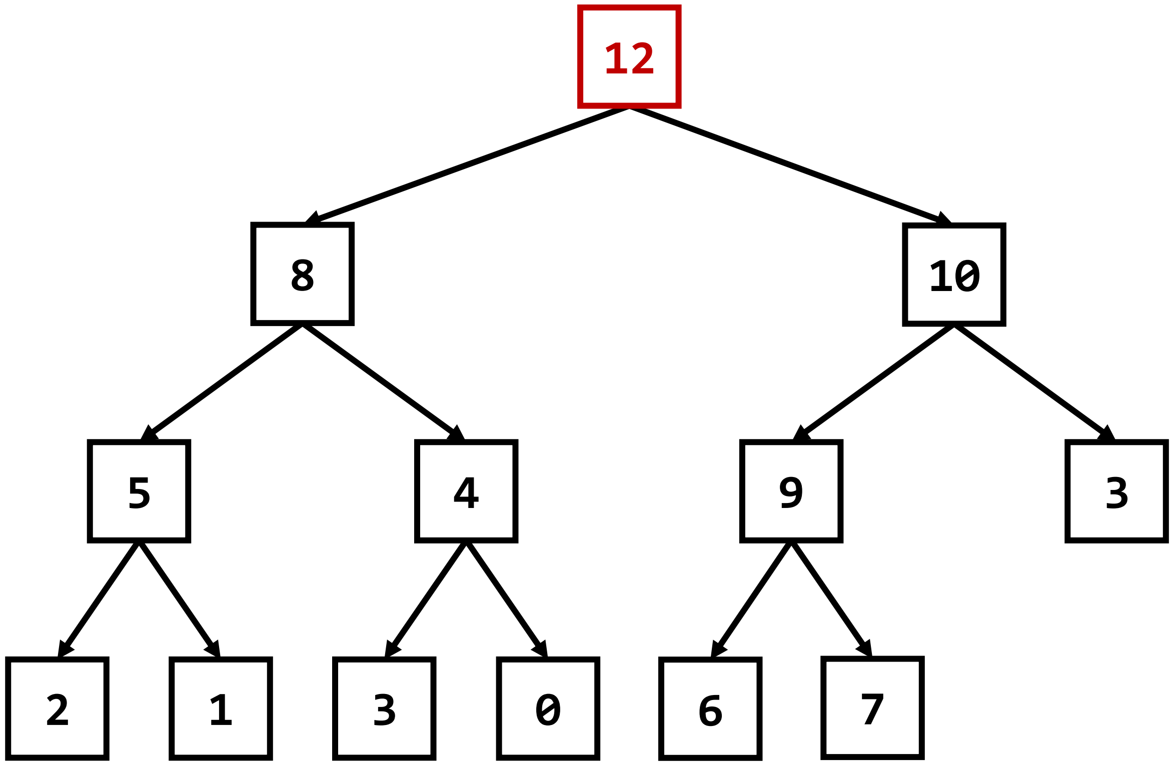
Examples



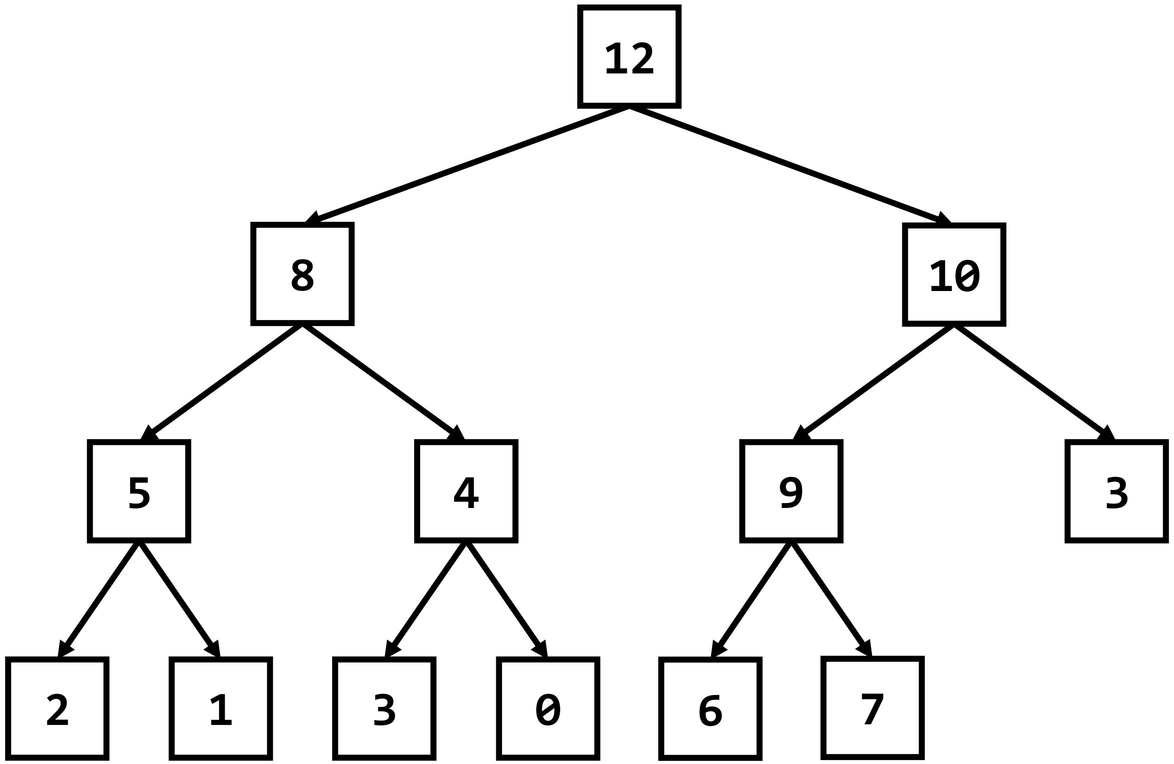
Examples



Examples

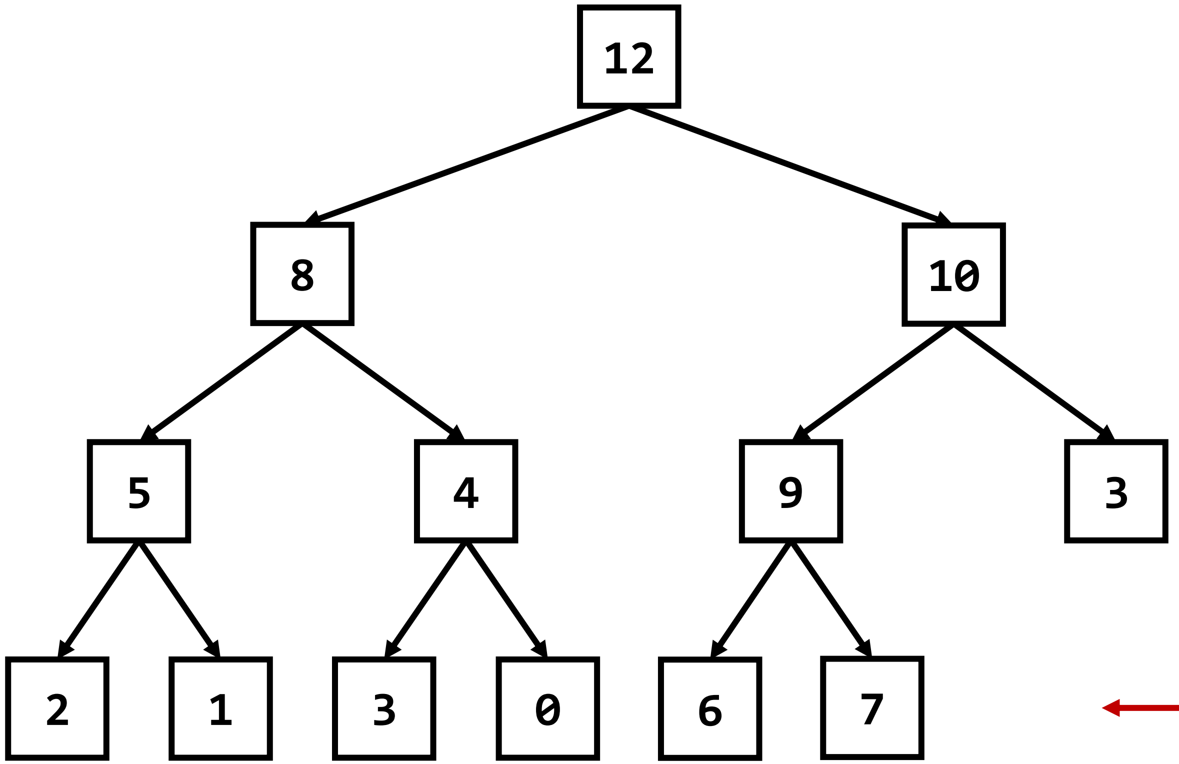


Examples



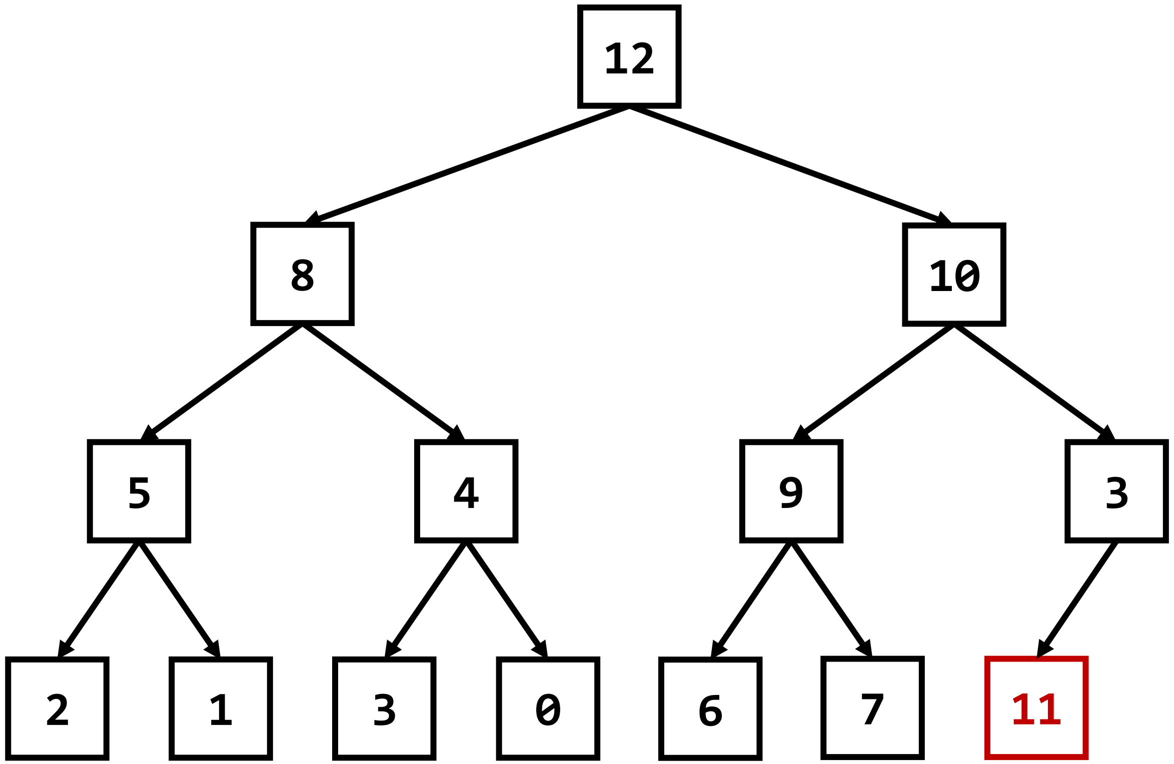
Done

Examples

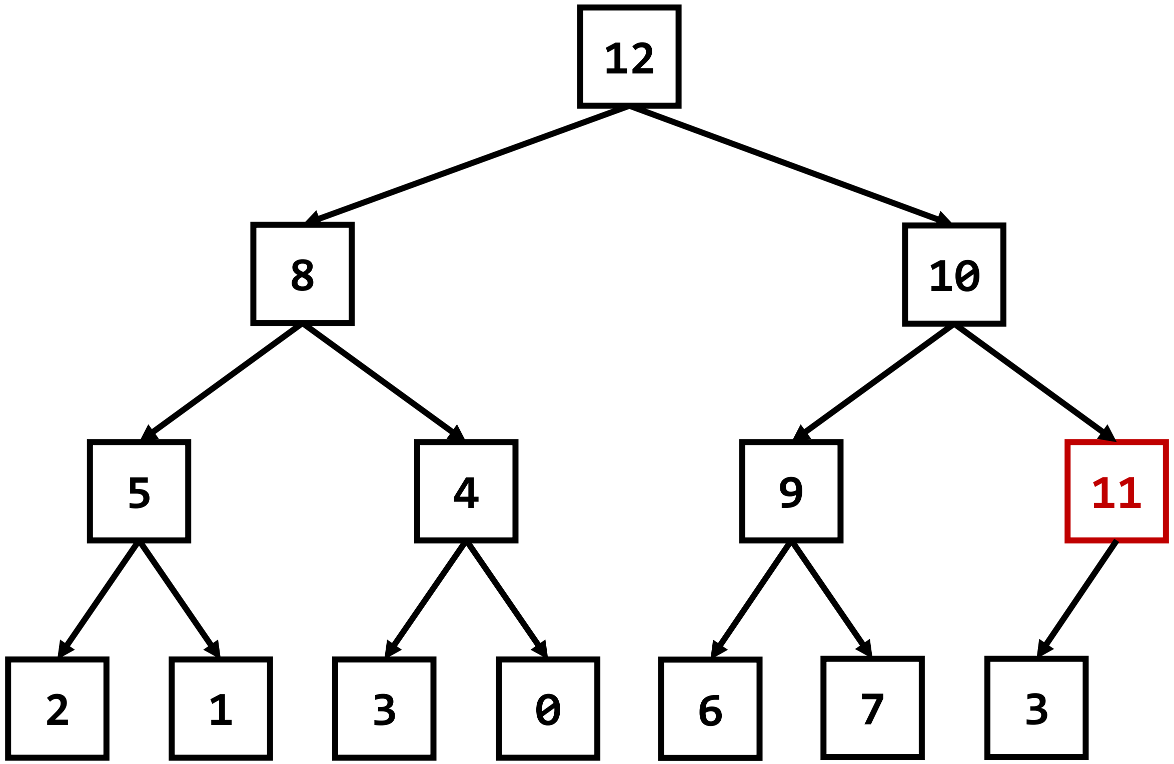


Insert 11

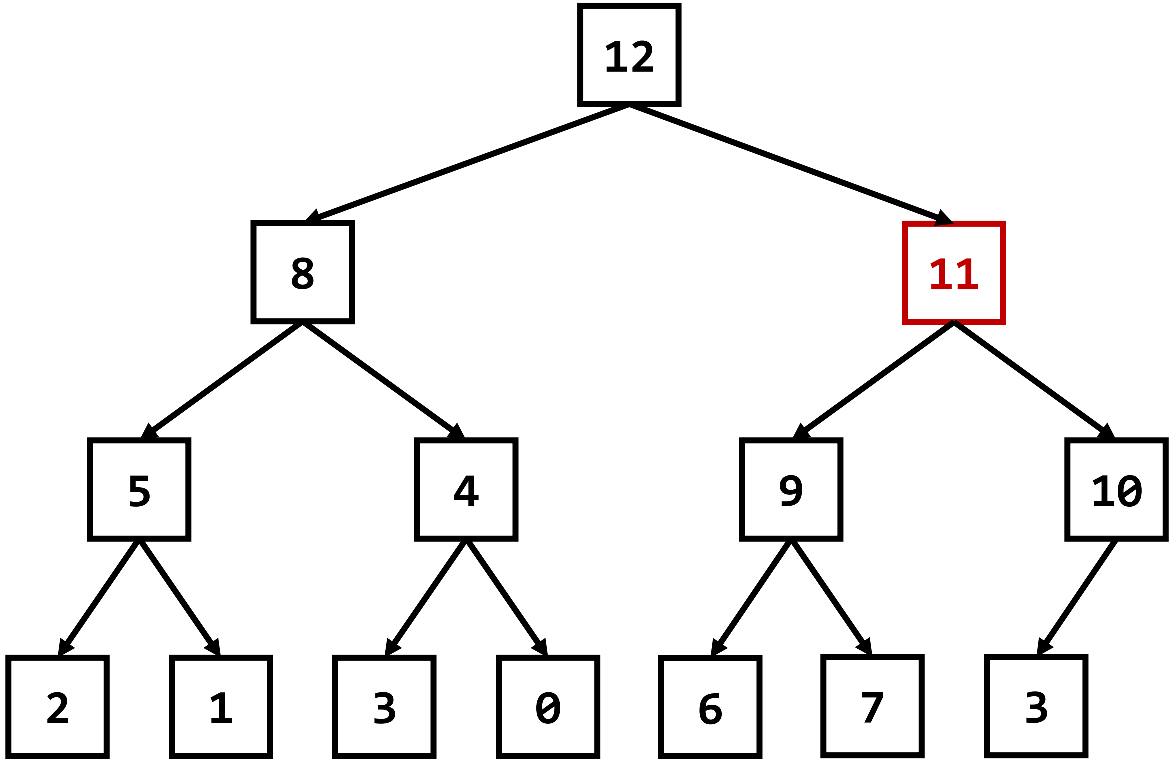
Examples



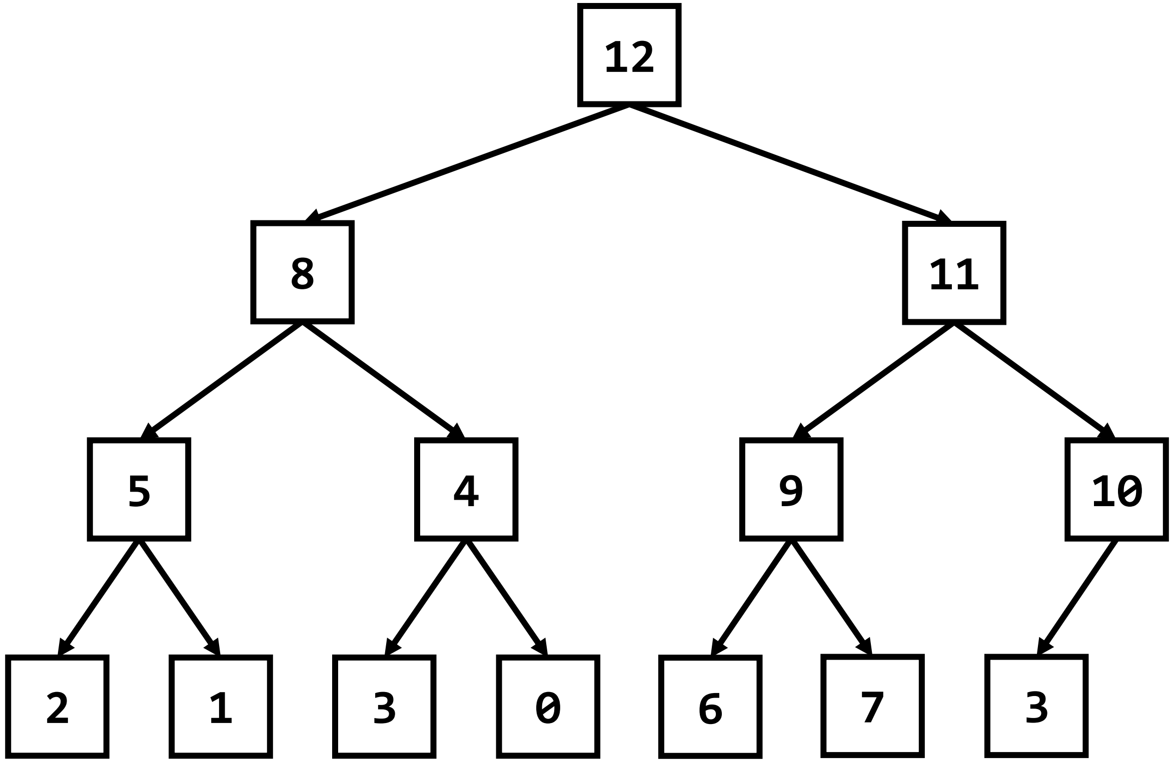
Examples



Examples

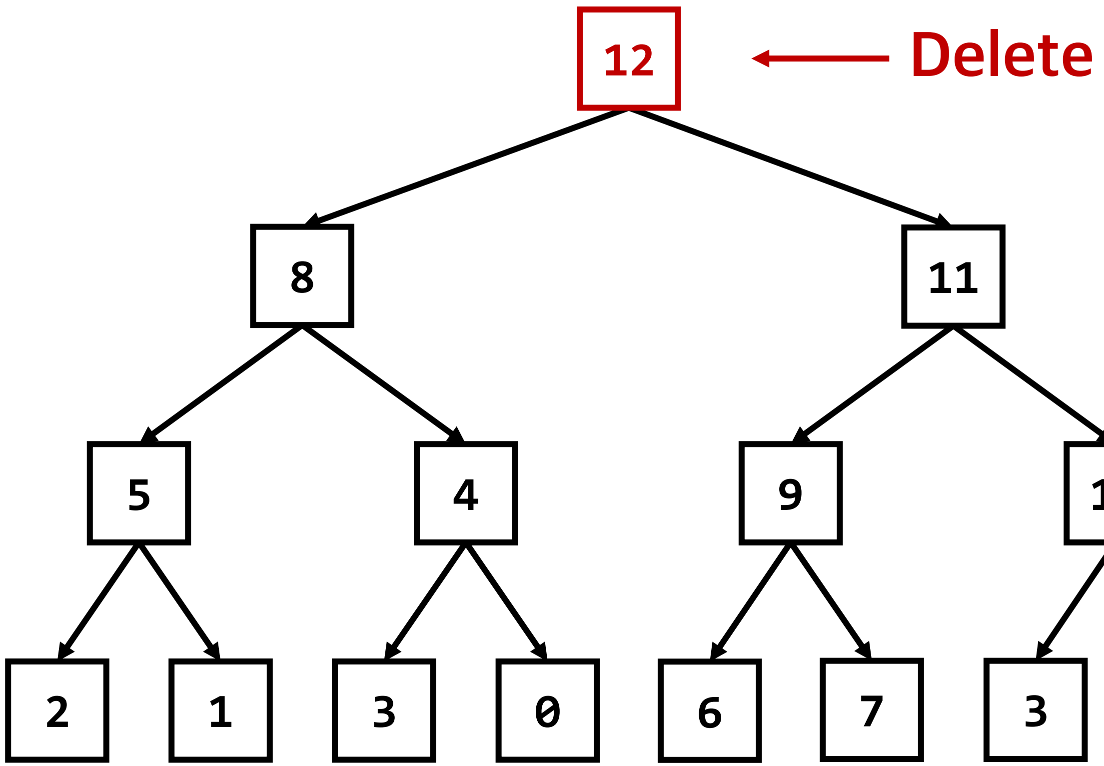


Examples

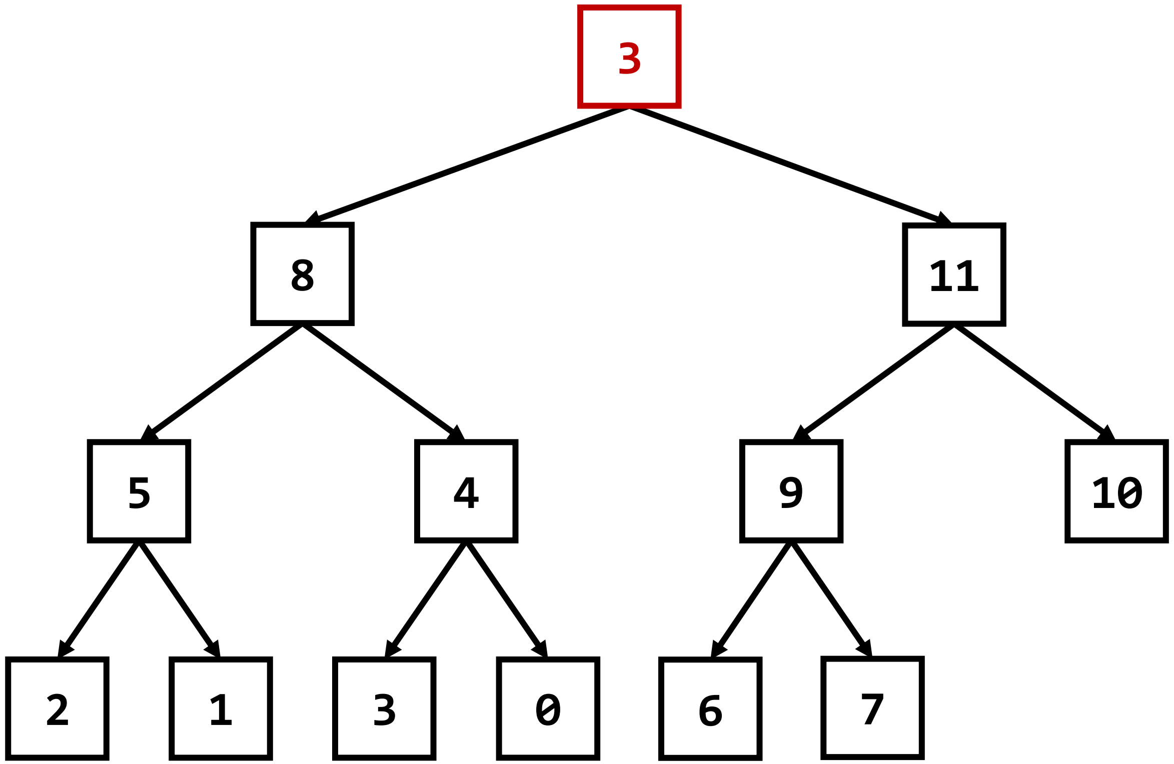


Done

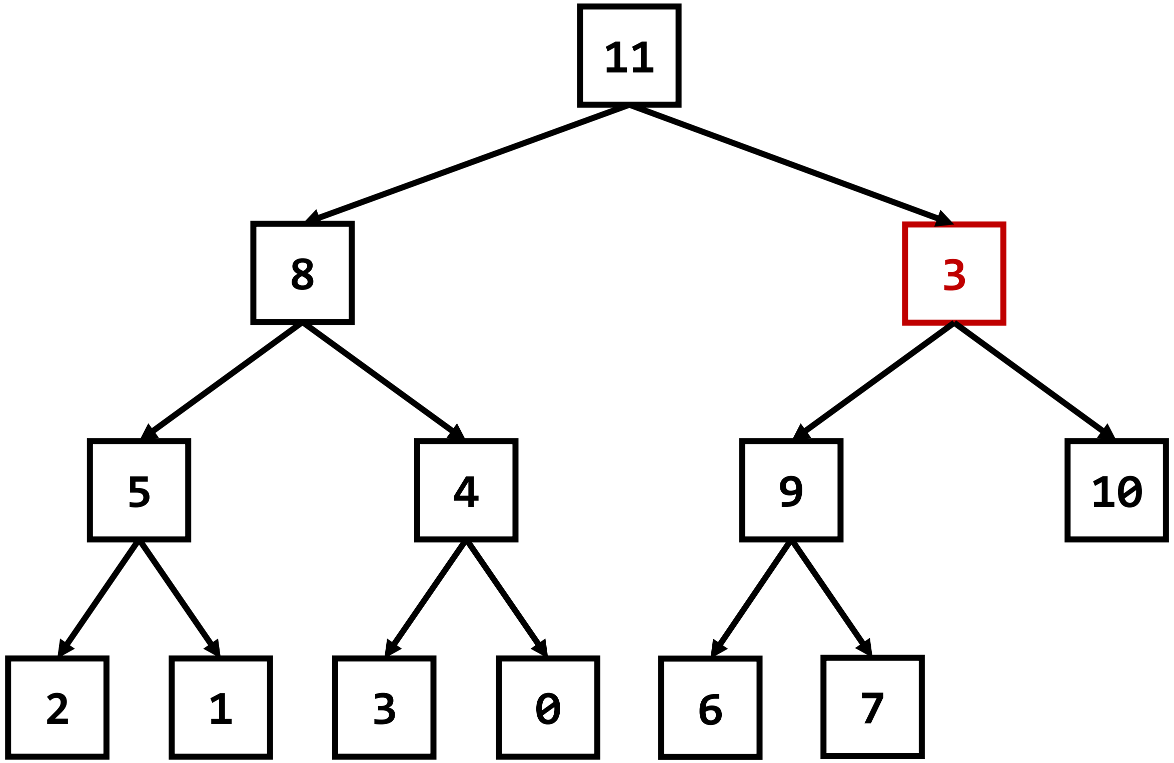
Examples



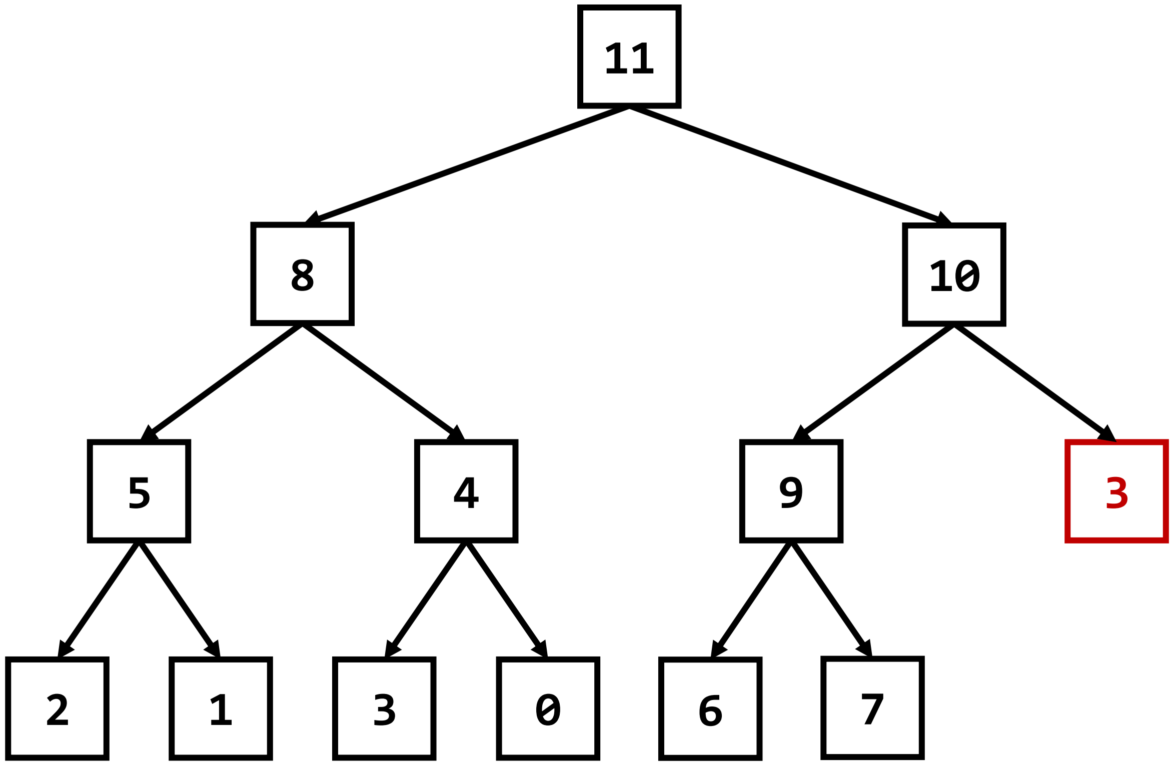
Examples



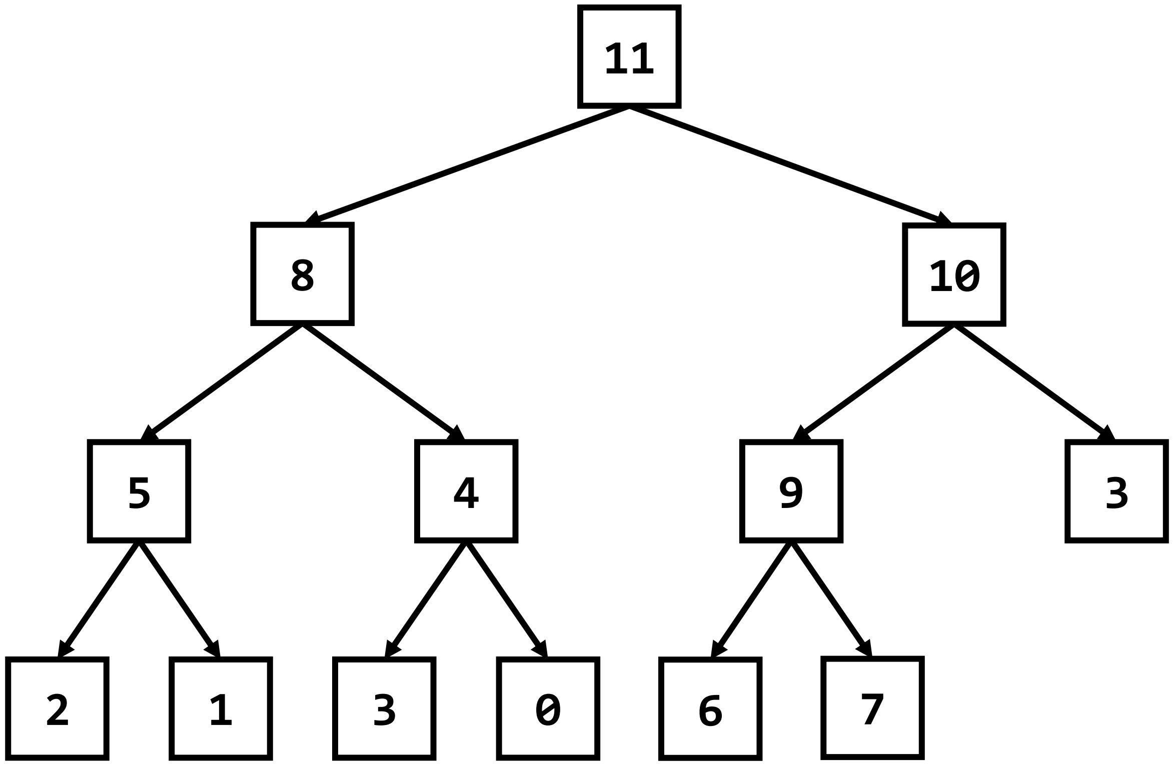
Examples



Examples



Examples



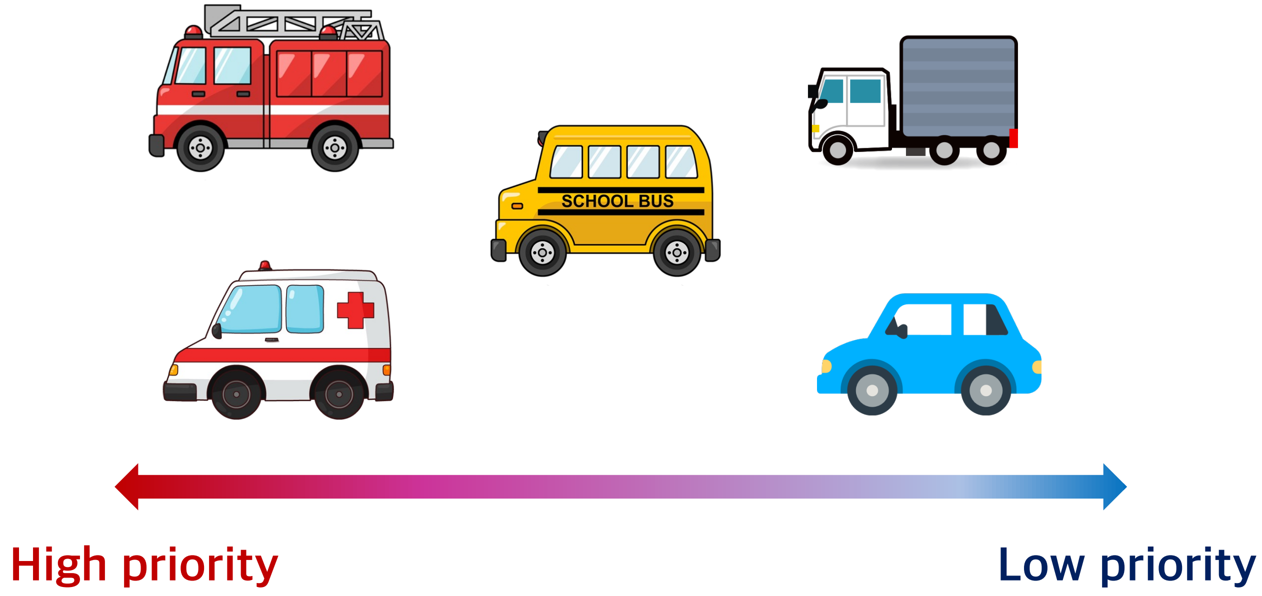
Done

Applications: Priority Queue

- What is the **priority queue**?
  - This queue does not follow the first-in first-out (FIFO) principle

Applications: Priority Queue

- What is the **priority queue**?
  - This queue does not follow the first-in first-out (FIFO) principle
  - Each element in the queue has its own priority
  - The most important element (i.e., highest priority) should come out first



Applications: Priority Queue

- What is the **priority queue**?
  - This queue does not follow the first-in first-out (FIFO) principle
  - Each element in the queue has its own priority
  - The most important element (i.e., highest priority) should come out first
- Priority queue operations
  - `enqueue()` - insert an element into the queue
  - `dequeue()` - delete the most important element from the queue
  - `peek()` - return the value of the most important element
  - These functions can be easily implemented using heap

Applications: Heap Sort

- What is **sorting**?
  - Sorting is the process of arranging elements in a specific order
  - Example: Sort below numbers in the increasing order

5	6	1	3	8	2	7	4

Sorting

1	2	3	4	5	6	7	8

Applications: Heap Sort

- What is **sorting**?
  - Sorting is the process of arranging elements in a specific order
  - Example: Sort below numbers in the increasing order

5	6	1	3	8	2	7	4

1. Insert all elements

- It is easily implemented using … **Heap**

2. Delete all elements from the heap

1	2	3	4	5	6	7	8

Time Complexities

- In the heap structure,
  - The height is $O(\log N)$ where N is the number of nodes
  - Insertion of a new node has $O(\log N)$ time complexity
  - Deletion of the root node has $O(\log N)$ time complexity

Time Complexities

- In the heap structure,
  - The height is $O(\log N)$ where N is the number of nodes
  - Insertion of a new node has $O(\log N)$ time complexity
  - Deletion of the root node has $O(\log N)$ time complexity

- In the priority queue,
  - Enqueue ($\approx$ insertion) has $O(\log N)$ time complexity
  - Dequeue ($\approx$ deletion) has $O(\log N)$ time complexity

Time Complexities

- In the heap structure,
  - The height is $O(\log N)$ where N is the number of nodes
  - Insertion of a new node has $O(\log N)$ time complexity
  - Deletion of the root node has $O(\log N)$ time complexity

- In the priority queue,
  - Enqueue ($\approx$ insertion) has $O(\log N)$ time complexity
  - Dequeue ($\approx$ deletion) has $O(\log N)$ time complexity

- In heap sort,
  - Sorting N elements requires $c \cdot (\log 1 + \log 2 + \cdots + \log N) = O(N \log N)$

Any Questions?