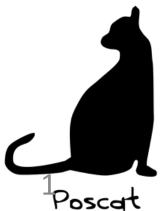
POSCAT Seminar 3-2 : Data Structure

yougatup @ POSCAT



Topic

- Topic today
  - Data Structure
    - Stack
    - Queue
    - ~~Linked List~~
    - Tree
    - Graph (definition)
    - Priority Queue
    - Heap

No slide

- More topic for you
  - Advanced DS
    - Binary Indexed Tree
    - Fenwick Tree
  - Well-known Problem
    - Parenthesis Matching
    - Range Minimum Query
    - Lowest Common Ancestor
    - Inversion Count

Stack

- LIFO (Last-In-First-Out)
  - Recursion 을 할 때 쓰이는 자료구조
  - 실제로 유용하게 쓸 일은 많지 않습니다
- Parenthesis Matching
  - 괄호의 짝을 서로 찾아주는 방법
  - Stack을 잘 쓰면 됩니다

(	(	)	(	(	)	)	(	)	)
5	1	1	3	2	2	3	4	4	5

- Depth First Search on Graph
  - Graph Traversal

Parenthesis Matching

- Use Stack !
  - "나"와 "나의 짝" 사이에는 완전한 괄호가 있어야 함
  - '('은 Push, ')' 은 Pop

index	0	1	2	3	4	5	6	7	8	9
	(	(	)	(	(	)	)	(	)	)

(	0

Parenthesis Matching

- Use Stack !
  - “나”와 “나의 짝” 사이에는 완전한 괄호가 있어야 함
  - ‘(‘은 Push, ‘)’ 은 Pop

index	0	1	2	3	4	5	6	7	8	9
	(	(	)	(	(	)	)	(	)	)

(	1
(	0

Parenthesis Matching

- Use Stack !
  - “나”와 “나의 짝” 사이에는 완전한 괄호가 있어야 함
  - ‘(‘은 Push, ‘)’ 은 Pop

index	0	1	2	3	4	5	6	7	8	9
	(	(	)	(	(	)	)	(	)	)
		1	1							

It is my partner !

(	1
(	0

Parenthesis Matching

- Use Stack !
  - "나"와 "나의 짝" 사이에는 완전한 괄호가 있어야 함
  - '('은 Push, ')' 은 Pop

index	0	1	2	3	4	5	6	7	8	9
	(	(	)	(	(	)	)	(	)	)
		1	1							

Pop!

(	0

Parenthesis Matching

- Use Stack !
  - “나”와 “나의 짝” 사이에는 완전한 괄호가 있어야 함
  - ‘(‘은 Push, ‘)’ 은 Pop

index	0	1	2	3	4	5	6	7	8	9
	(	(	)	(	(	)	)	(	)	)
		1	1							

(	3
(	0

Parenthesis Matching

- Use Stack !
  - "나"와 "나의 짝" 사이에는 완전한 괄호가 있어야 함
  - '('은 Push, ')' 은 Pop

index	0	1	2	3	4	5	6	7	8	9
	(	(	)	(	(	)	)	(	)	)
		1	1							

(	4
(	3
(	0

Parenthesis Matching

- Use Stack !
  - "나"와 "나의 짝" 사이에는 완전한 괄호가 있어야 함
  - '('은 Push, ')' 은 Pop

index	0	1	2	3	4	5	6	7	8	9
	(	(	)	(	(	)	)	(	)	)
		1	1		2	2				

It is my partner !

(	4
(	3
(	0

Parenthesis Matching

- Use Stack !
  - "나"와 "나의 짝" 사이에는 완전한 괄호가 있어야 함
  - '('은 Push, ')' 은 Pop

index	0	1	2	3	4	5	6	7	8	9
	(	(	)	(	(	)	)	(	)	)
		1	1		2	2				

Pop!

(	3
(	0

Parenthesis Matching

- Use Stack !
  - “나”와 “나의 짝” 사이에는 완전한 괄호가 있어야 함
  - ‘(‘은 Push, ‘)’ 은 Pop

index	0	1	2	3	4	5	6	7	8	9
	(	(	)	(	(	)	)	(	)	)
		1	1	3	2	2	3			

It is my partner !

(	3
(	0

Parenthesis Matching

- Use Stack !
  - "나"와 "나의 짝" 사이에는 완전한 괄호가 있어야 함
  - '('은 Push, ')' 은 Pop

index	0	1	2	3	4	5	6	7	8	9
	(	(	)	(	(	)	)	(	)	)
		1	1	3	2	2	3			

Pop!

(	0

Parenthesis Matching

- Use Stack !
  - "나"와 "나의 짝" 사이에는 완전한 괄호가 있어야 함
  - '('은 Push, ')' 은 Pop

index	0	1	2	3	4	5	6	7	8	9
	(	(	)	(	(	)	)	(	)	)
		1	1	3	2	2	3			

(	7
(	0

Parenthesis Matching

- Use Stack !
  - "나"와 "나의 짝" 사이에는 완전한 괄호가 있어야 함
  - '('은 Push, ')' 은 Pop

index	0	1	2	3	4	5	6	7	8	9
	(	(	)	(	(	)	)	(	)	)
		1	1	3	2	2	3	4	4	

It is my partner !

(	7
(	0

Parenthesis Matching

- Use Stack !
  - "나"와 "나의 짝" 사이에는 완전한 괄호가 있어야 함
  - '('은 Push, ')' 은 Pop

index	0	1	2	3	4	5	6	7	8	9
	(	(	)	(	(	)	)	(	)	)
		1	1	3	2	2	3	4	4	

Pop!

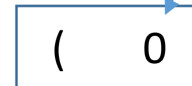
(	0

Parenthesis Matching

- Use Stack !
  - "나"와 "나의 짝" 사이에는 완전한 괄호가 있어야 함
  - '('은 Push, ')' 은 Pop

index	0	1	2	3	4	5	6	7	8	9
	(	(	)	(	(	)	)	(	)	)
	5	1	1	3	2	2	3	4	4	5

It is my partner !



Parenthesis Matching

- Use Stack !
  - "나"와 "나의 짝" 사이에는 완전한 괄호가 있어야 함
  - '('은 Push, ')' 은 Pop

index	0	1	2	3	4	5	6	7	8	9
	(	(	)	(	(	)	)	(	)	)
	5	1	1	3	2	2	3	4	4	5

Pop!

Parenthesis Matching

- Use Stack !
  - "나"와 "나의 짝" 사이에는 완전한 괄호가 있어야 함
  - '('은 Push, ')' 은 Pop
- Invalid Parenthesis Check
  - Pop을 해야 하는 상황에 Pop할 것이 없다면 ? → Invalid !
  - 최종 Stack에 괄호가 남아있는 경우? → Also Invalid !

(( ))) (( )))

We can't pop because stack it empty

(( ) (( )) ( )

After all operations, stack is not empty

Parenthesis Matching

- Use Stack !
  - "나"와 "나의 짝" 사이에는 완전한 괄호가 있어야 함
  - '('은 Push, ')' 은 Pop
- Invalid Parenthesis Check
  - Pop을 해야 하는 상황에 Pop할 것이 없다면 ? → Invalid !
  - 최종 Stack에 괄호가 남아있는 경우? → Also Invalid !

Verification : Is this algorithm **true** ? Think about it

Efficiency : What time does it **take** ? O(n)

Queue

- FIFO (First – In – First – Out)
  - 먼저 들어간 애가 먼저 나옴
  - Front, Rear pointer
  - 아직은 쓸 일 없습니다
  - **Circular Queue**
- BFS (Breadth First Search on Graph)
  - Graph Traversal

Tree

- Cycle 이 없는 Graph
  - Do you know what is Graph ?
  - 두 노드 사이의 경로는 유일하다 **// Why ?**
- Binary Tree
  - 자식 노드가 2개 이하인 Tree
  - Binary Tree Traversal
- Complete Binary Tree
  - 자식 노드가 왼쪽부터 채워지는 Binary Tree
  - Heap
- Full Binary Tree
  - Leaf node 를 제외한 모든 Internal node의 자식이 2개
  - Binary Indexed Tree

Binary Tree

- Tree Traversal
  - Preorder → Root – Left – Right
  - Inorder → Left – Root – Right
  - Postorder → Left – Right – Root
- Preorder와 Inorder만으로 Tree를 유일하게 결정

  Preorder : 1 2 4 3 5 6 7
  Inorder : 4 2 1 5 3 7 6 일 때 Tree는 어떻게 생겼나 ?

Binary Tree

- Preorder와 Inorder만으로 Tree를 유일하게 결정

  Preorder : 1 2 4 3 5 6 7
  Inorder : 4 2 1 5 3 7 6 일 때 Tree는 어떻게 생겼나 ?

1. Preorder의 가장 첫 번째 node는 Tree의 root 이다.
2. Root node r에 대하여 Inorder의 순서를 고려하였을 때,
   r의 왼쪽에는 r의 left-subtree의 node들이,
   오른쪽에는 right-subrree의 node들이 있다

Binary Tree

Root!

- Preorder와 Inorder만으로 Tree를 유일하게 결정

Preorder : **1** 2 4 3 5 6 7

Inorder : 4 2 **1** 5 3 7 6 일 때 Tree는 어떻게 생겼나 ?

1

Binary Tree

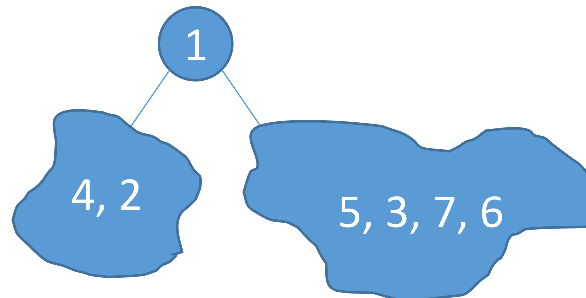
Root!

- Preorder와 Inorder만으로 Tree를 유일하게 결정

Preorder : **1** 2 4 3 5 6 7

Inorder : 4 2 **1** 5 3 7 6 일 때 Tree는 어떻게 생겼나 ?

left right

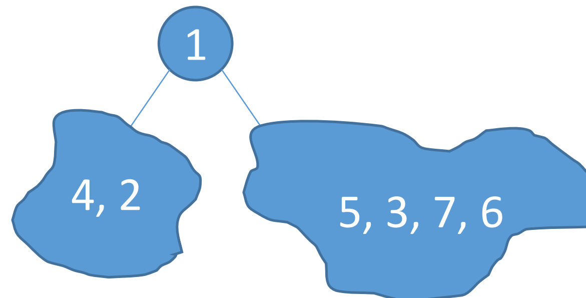


Binary Tree

Divide & Conquer !

- Preorder와 Inorder만으로 Tree를 유일하게 결정

Preorder : 1 2 4 3 5 6 7
Inorder : 4 2 1 5 3 7 6 일 때 Tree는 어떻게 생겼나 ?
left right



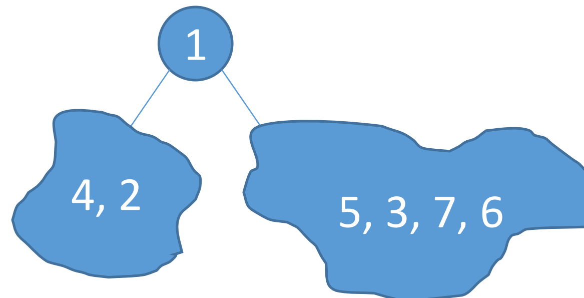
Binary Tree

Divide & Conquer !

- Preorder와 Inorder만으로 Tree를 유일하게 결정

Preorder : 1 2 4 3 5 6 7
Inorder : 4 2 1 5 3 7 6 일 때 Tree는 어떻게 생겼나 ?
left right

Preorder : 2 4
Inorder : 4 2



Preorder : 3 5 6 7
Inorder : 5 3 7 6

Binary Tree

Divide & Conquer !

- Preorder와 Inorder만으로 Tree를 유일하게 결정

Preorder : 1 2 4 3 5 6 7

Inorder : 4 2 1 5 3 7 6 일 때 Tree는 어떻게 생겼나 ?

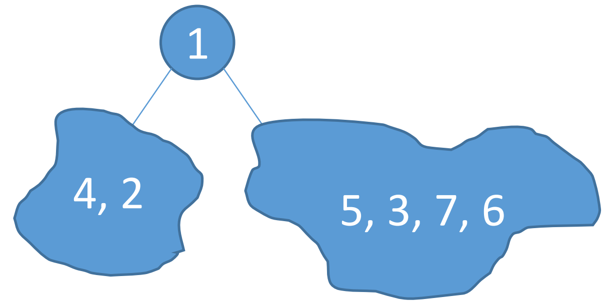
left right

Root !

Preorder : 2 4

Inorder : 4 2

left



Preorder : 3 5 6 7

Inorder : 5 3 7 6

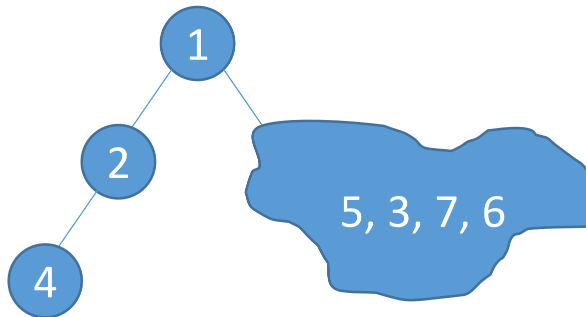
Binary Tree

Divide & Conquer !

- Preorder와 Inorder만으로 Tree를 유일하게 결정

Preorder : 1 2 4 3 5 6 7
Inorder : 4 2 1 5 3 7 6 일 때 Tree는 어떻게 생겼나 ?
left right

Root !
Preorder : 2 4
Inorder : 4 2
left



Preorder : 3 5 6 7
Inorder : 5 3 7 6

Binary Tree

Divide & Conquer !

- Preorder와 Inorder만으로 Tree를 유일하게 결정

Preorder : **1** 2 4 3 5 6 7

Inorder : 4 2 **1** 5 3 7 6 일 때 Tree는 어떻게 생겼나 ?

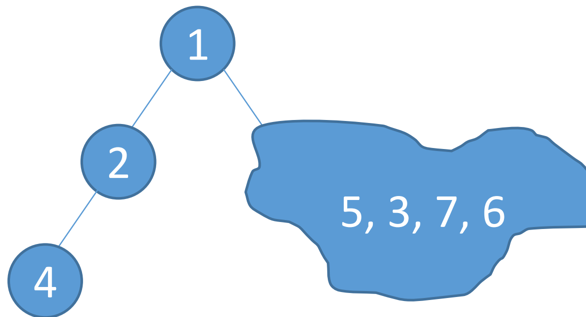
left right

Root !

Preorder : **2** 4

Inorder : 4 **2**

left



Preorder : **3** 5 6 7

Inorder : 5 **3** 7 6

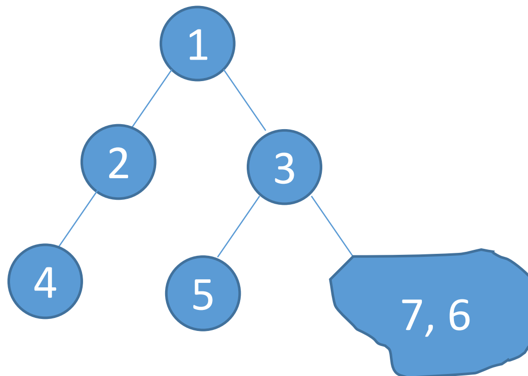
Binary Tree

Divide & Conquer !

- Preorder와 Inorder만으로 Tree를 유일하게 결정

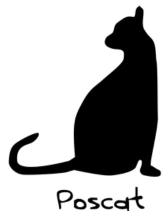
Preorder : 1 2 4 3 5 6 7
Inorder : 4 2 1 5 3 7 6 일 때 Tree는 어떻게 생겼나 ?
left right

Root !
Preorder : 2 4
Inorder : 4 2
left



Preorder : 3 5 6 7
Inorder : 5 3 7 6

Preorder : 6 7
Inorder : 7 6



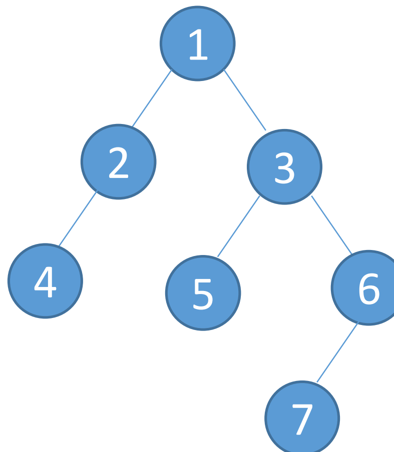
Binary Tree

Divide & Conquer !

- Preorder와 Inorder만으로 Tree를 유일하게 결정

Preorder : **1** 2 4 3 5 6 7
Inorder : 4 2 **1** 5 3 7 6 일 때 Tree는 어떻게 생겼나 ?
left right

Root !
Preorder : **2** 4
Inorder : 4 **2**
left



Preorder : **3** 5 6 7
Inorder : 5 **3** 7 6

Preorder : **6** 7
Inorder : 7 **6**

Priority Queue

- 우선순위가 있는 Queue
  - Pop을 하면, 우선순위가 가장 높은 element를 뺀다
  - How to implement it?

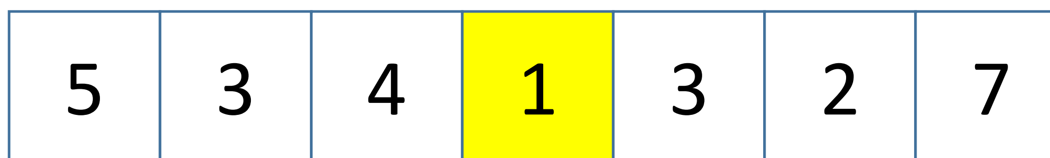
Priority Queue

- 우선순위가 있는 Queue
  - Pop을 하면, 우선순위가 가장 높은 element를 뺀다
  - How to implement it? **With array !**

5	3	4	1	3	2	7

Priority Queue

- 우선순위가 있는 Queue
  - Pop을 하면, 우선순위가 가장 높은 element를 뺀다
  - How to implement it? **With array !**



Pop !

Priority Queue

- 우선순위가 있는 Queue
  - Pop을 하면, 우선순위가 가장 높은 element를 뺀다
  - How to implement it? **With array !**

5	3	4		3	2	7

Priority Queue

- 우선순위가 있는 Queue
  - Pop을 하면, 우선순위가 가장 높은 element를 뺀다
  - How to implement it? **With array !**

Pop !

Priority Queue

- 우선순위가 있는 Queue
  - Pop을 하면, 우선순위가 가장 높은 element를 뺀다
  - How to implement it? **With array !**

Push 8

Priority Queue

- 우선순위가 있는 Queue
  - Pop을 하면, 우선순위가 가장 높은 element를 뺀다
  - How to implement it? **With array !**

5	3	4	8	3		7

Priority Queue

- 우선순위가 있는 Queue
  - Pop을 하면, 우선순위가 가장 높은 element를 뺀다
  - How to implement it? **With array !**

Verification : Is this algorithm **true** ?

Efficiency : What time does it **take** ?

Priority Queue

- 우선순위가 있는 Queue
  - Pop을 하면, 우선순위가 가장 높은 element를 뺀다
  - How to implement it? **With array !**

Verification : Is this algorithm **true** ? Trivial

Efficiency : What time does it **take** ? **O(n)** for push & pop

Priority Queue

- 우선순위가 있는 Queue
  - Pop을 하면, 우선순위가 가장 높은 element를 뺀다
  - How to implement it? **With array !**

Verification : Is this algorithm **true** ? Trivial

Efficiency : What time does it **take** ? **O(n)** for push & pop

Is there another algorithm faster ?

Priority Queue

- 우선순위가 있는 Queue
  - Pop을 하면, 우선순위가 가장 높은 element를 뺀다
  - How to implement it? **With array !**

Verification : Is this algorithm **true** ? Trivial

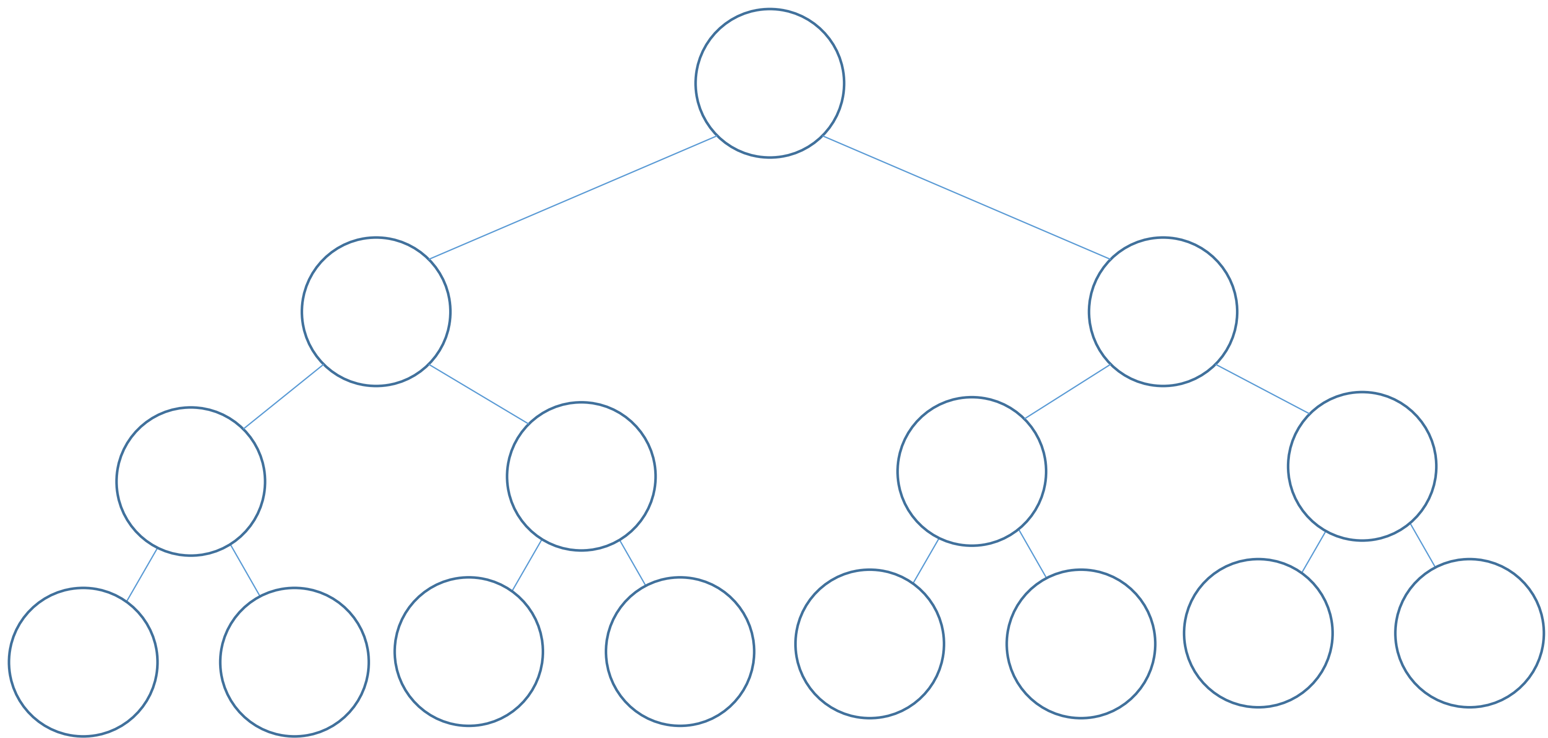
Efficiency : What time does it **take** ? **O(n)** for push & pop

Is there another algorithm faster ?
→ **Use heap !**

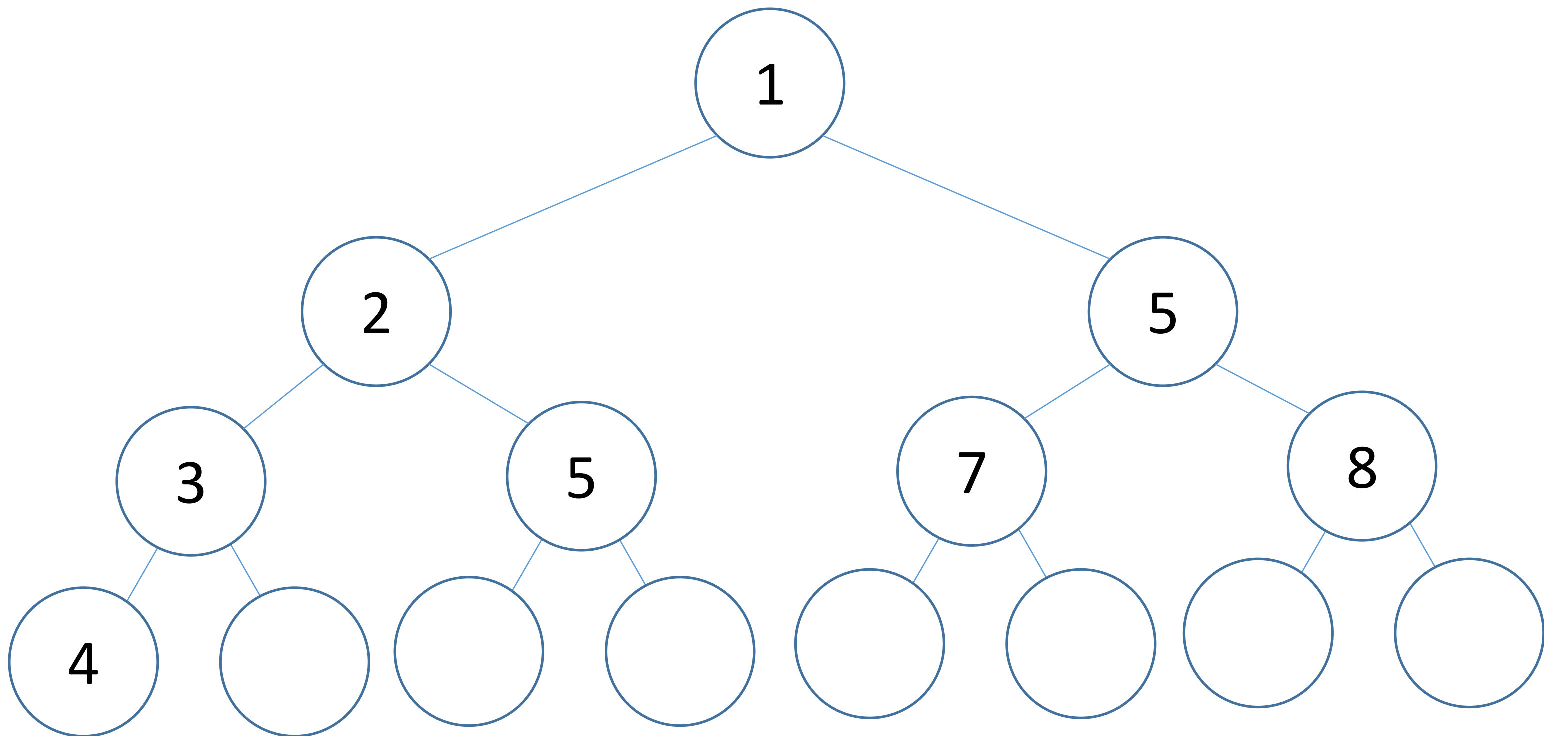
Heap

- Complete Binary Tree with Invarient
  - Complete Binary Tree
  - Inv : 부모노드의 값이 항상 자식 노드의 값보다 우선순위가 높음
- Priority
  - Greater First → Max heap
  - Less First → Min heap
- Operation
  - Push : Heap의 가장 끝에 추가시키고 Rearrange
  - Pop : Heap의 root를 빼고, Heap의 가장 끝에 있는 elemen를 root로 올린 후 Rearrange

Min Heap

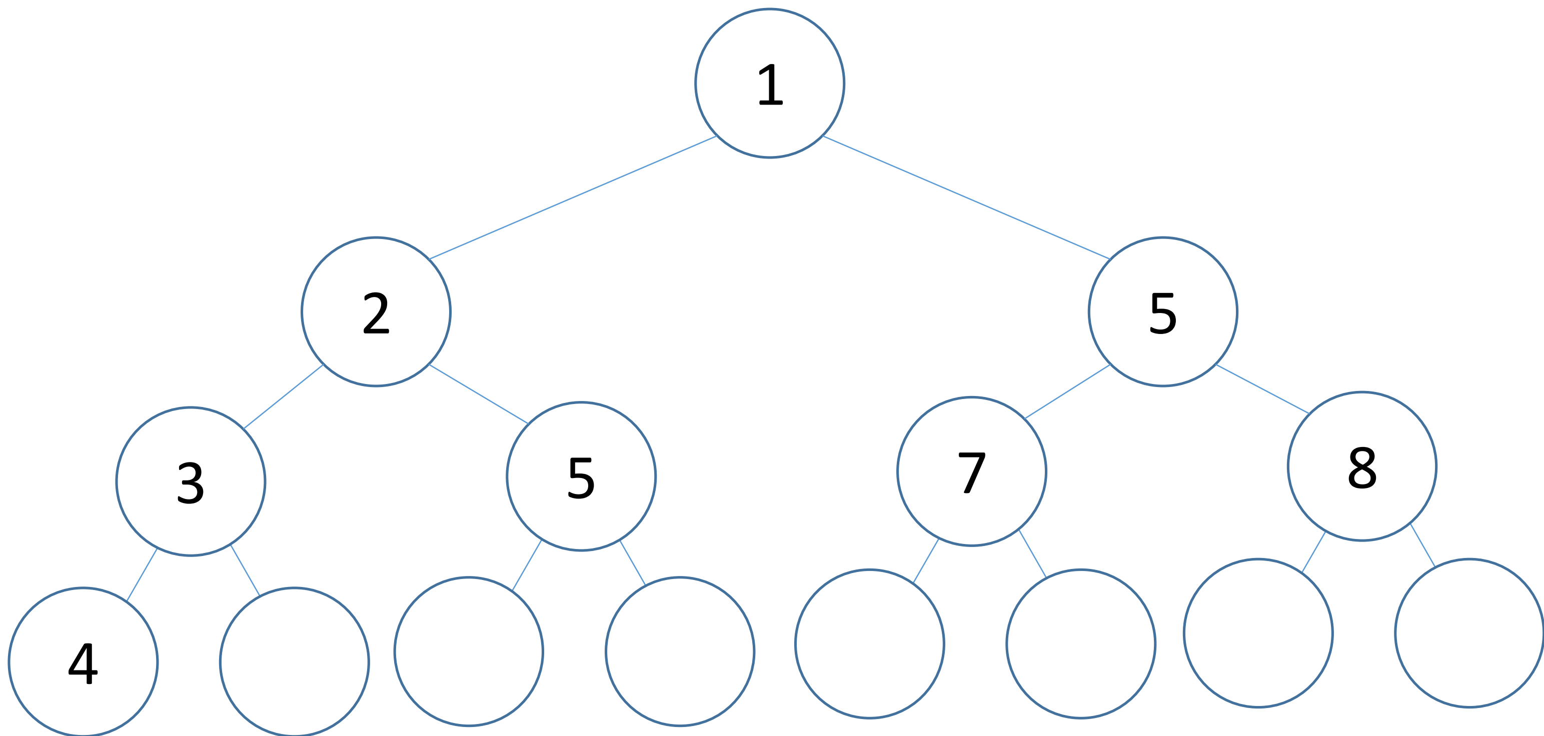


Min Heap



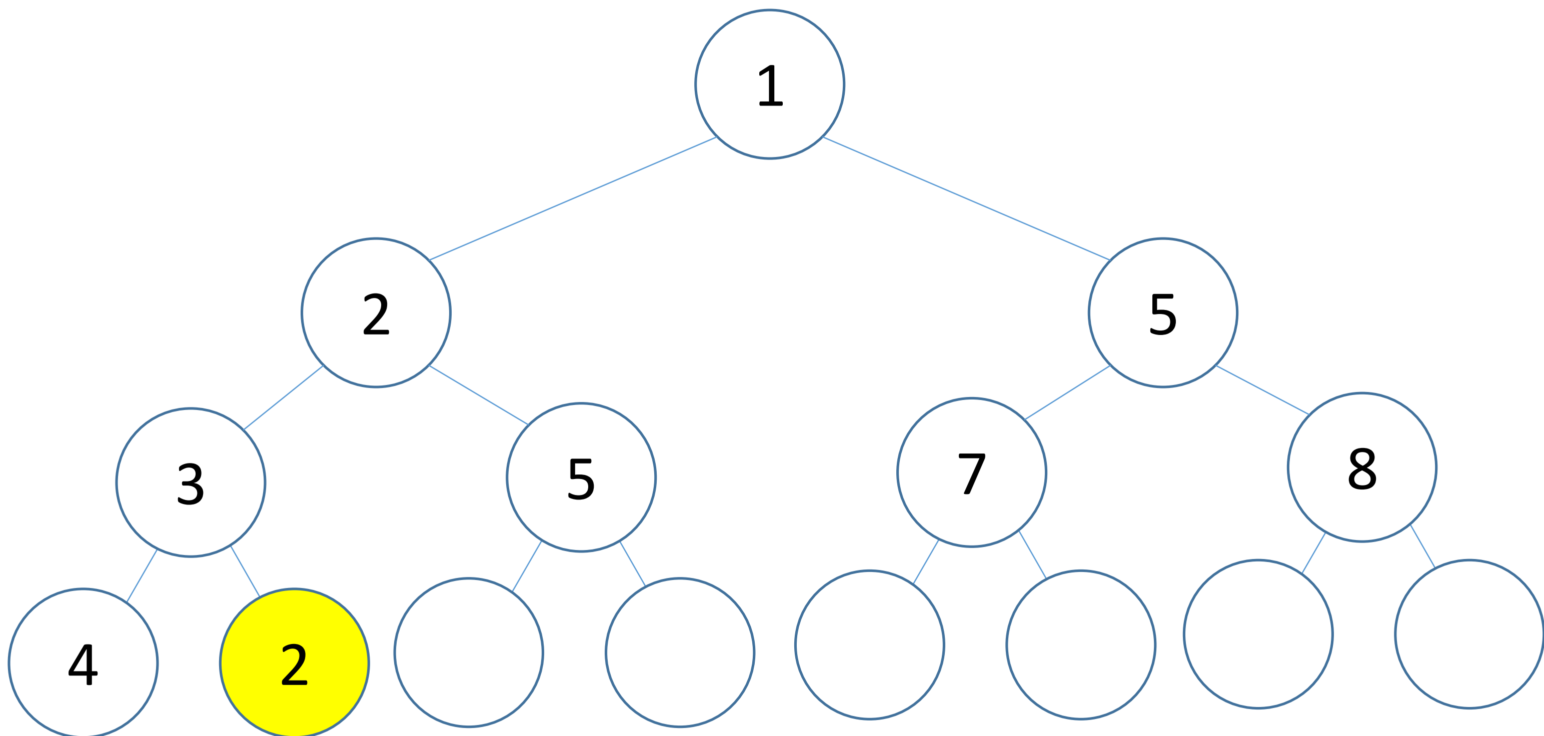
Min Heap

Push 2



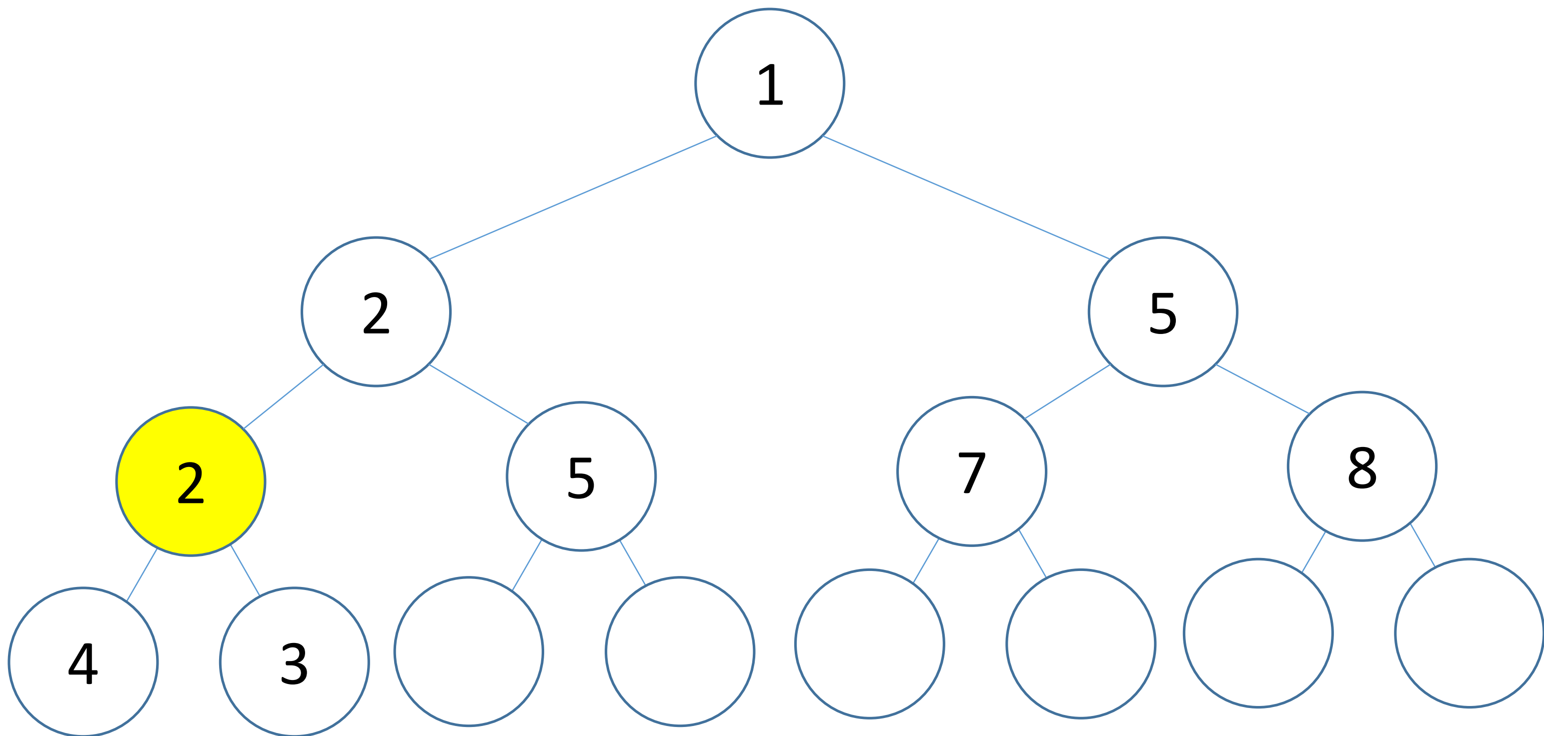
Min Heap

Push 2



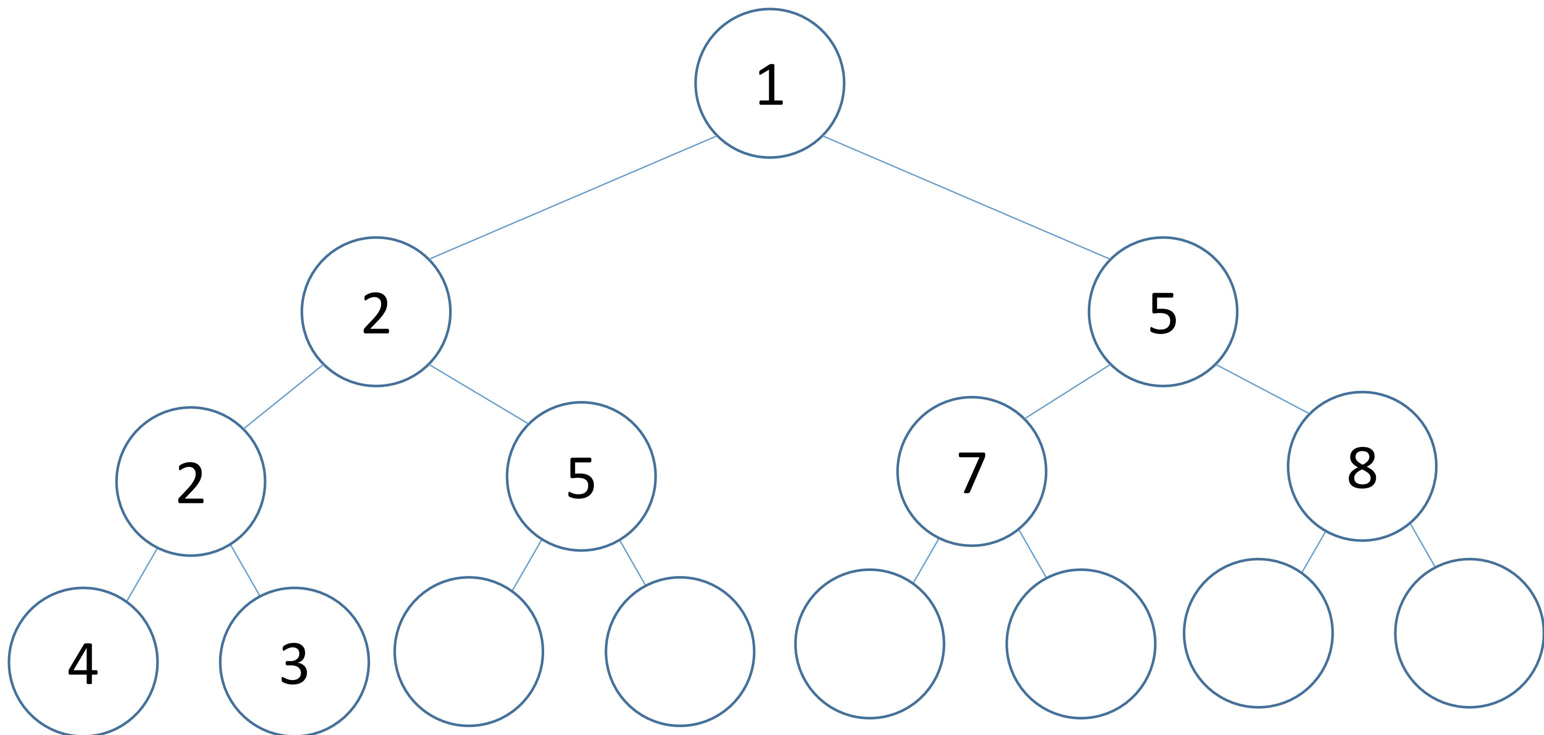
Min Heap

Push 2



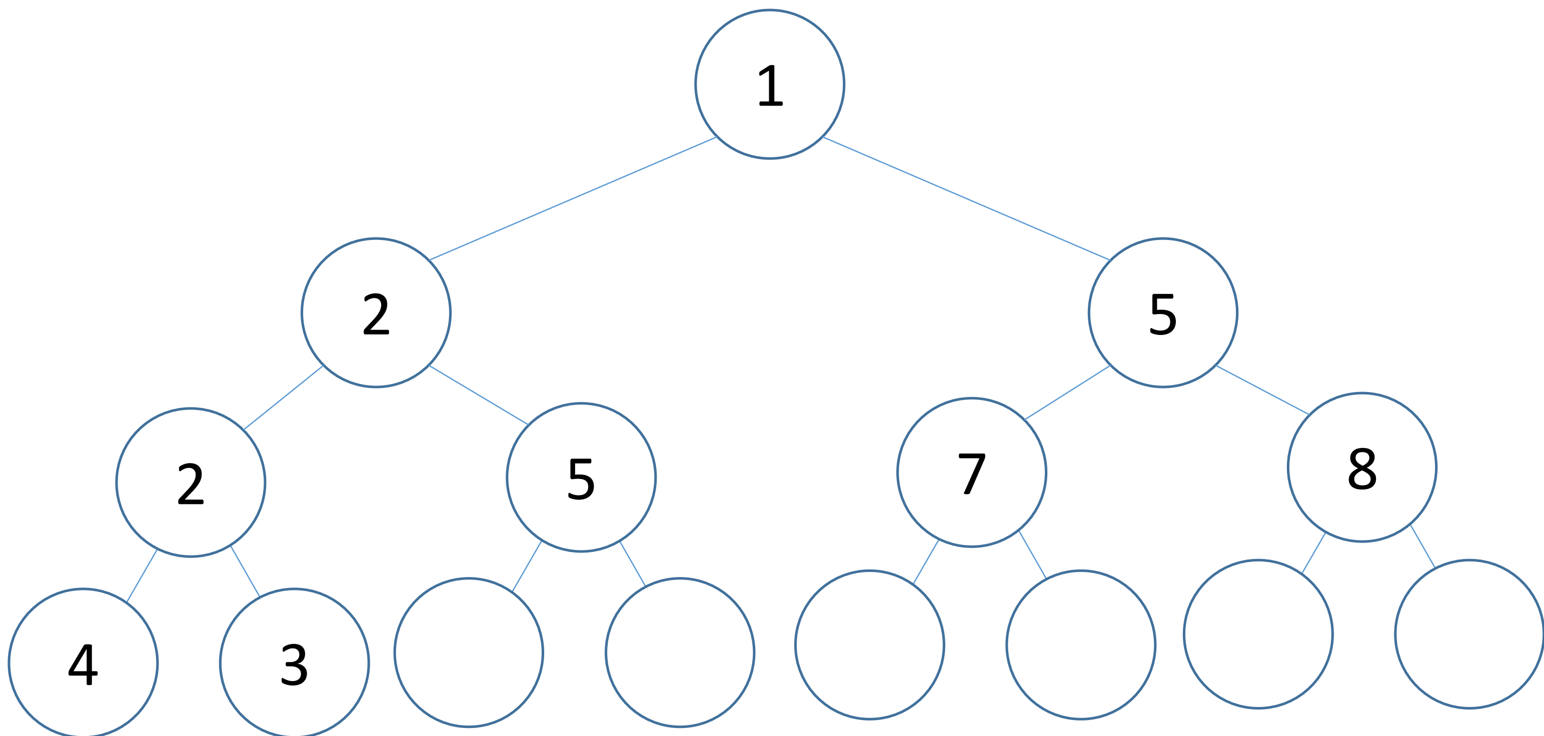
Min Heap

Push 2 // Done



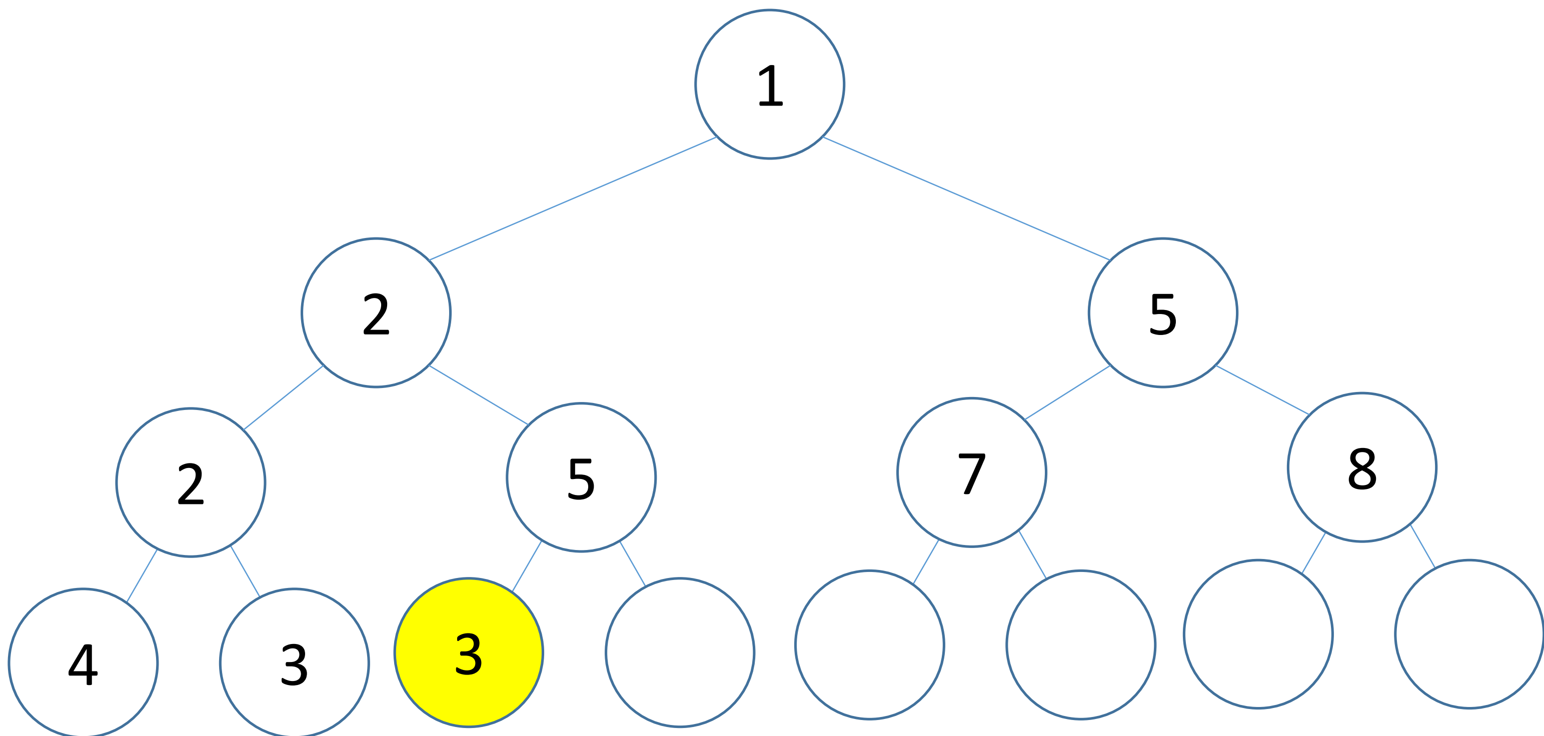
Min Heap

Push 3



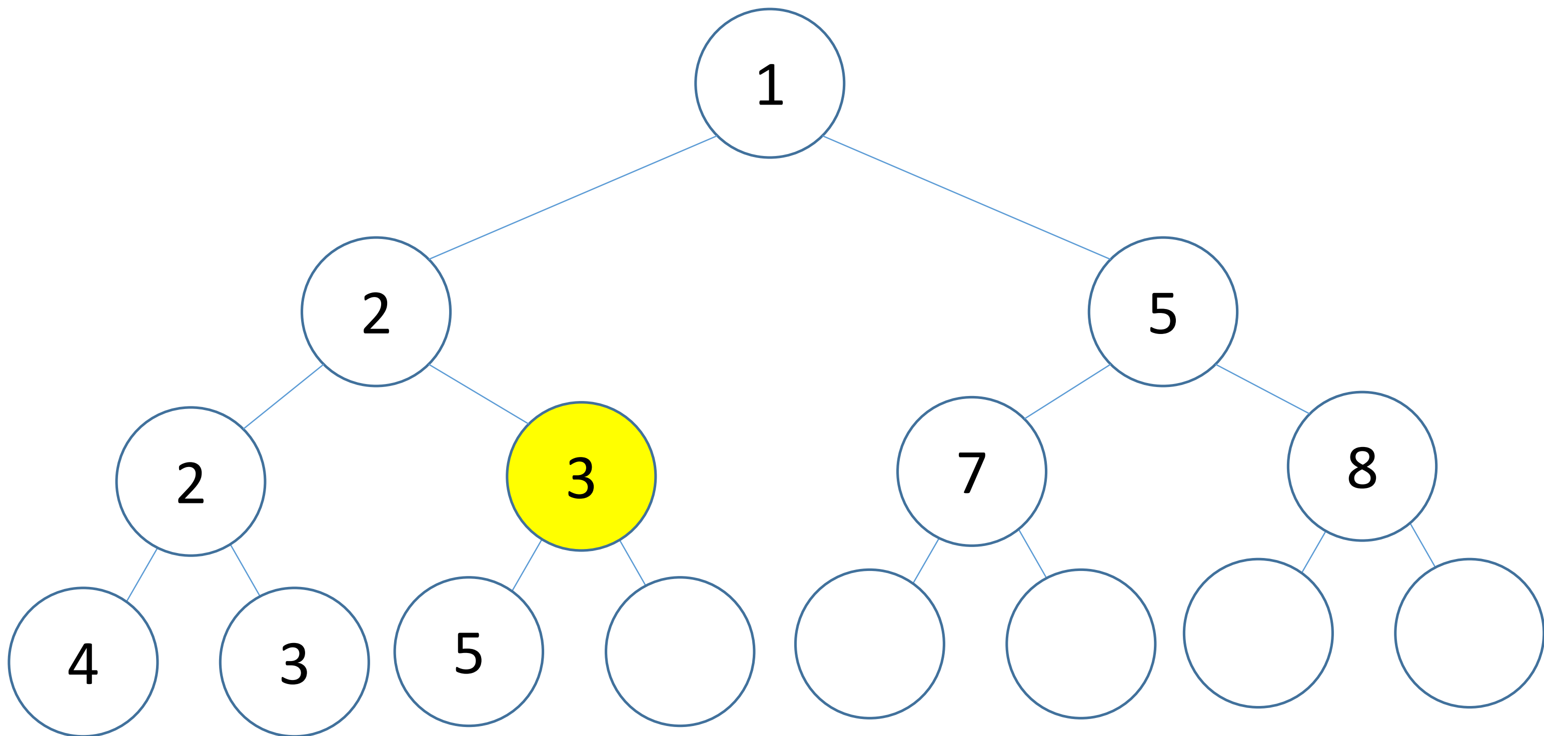
Min Heap

Push 3



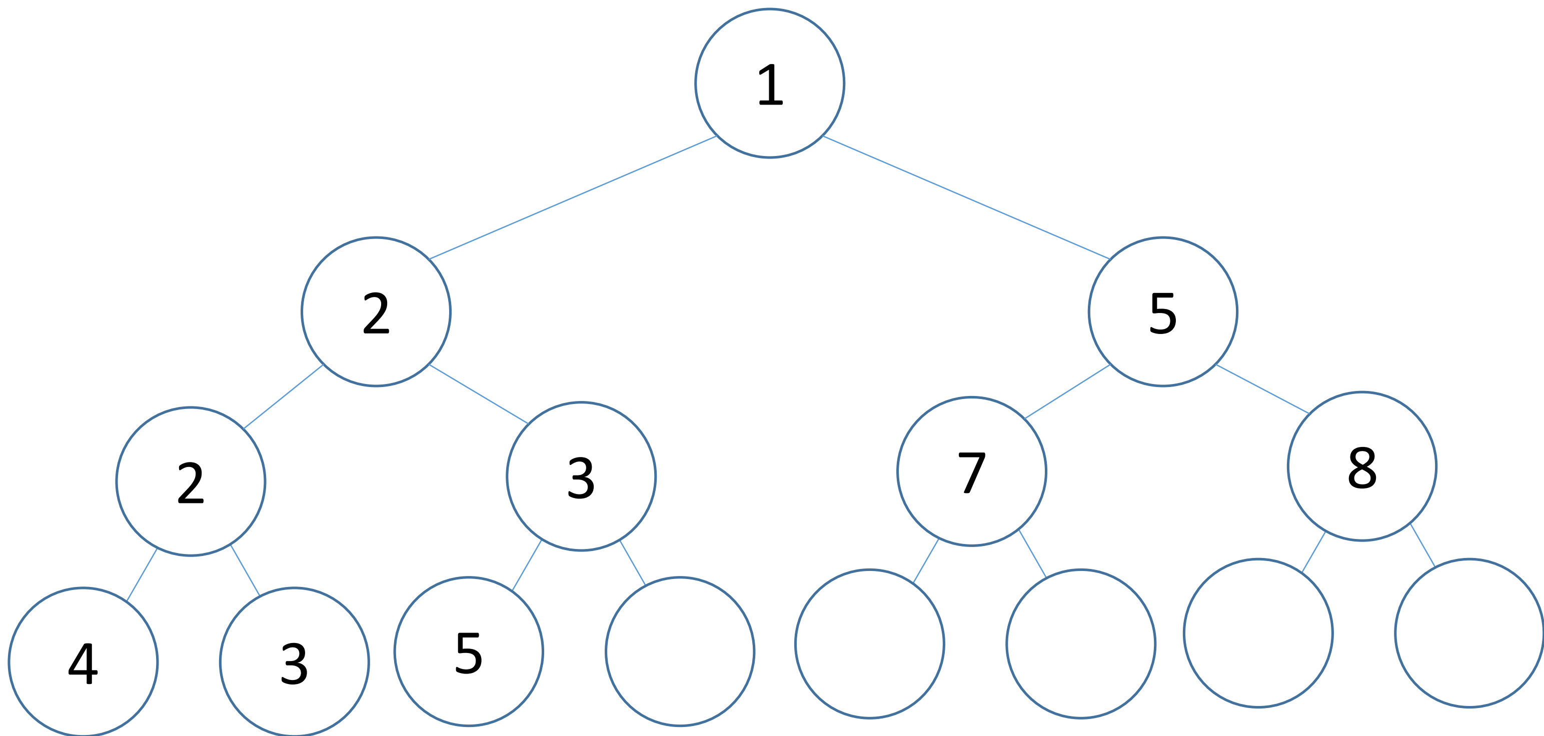
Min Heap

Push 3



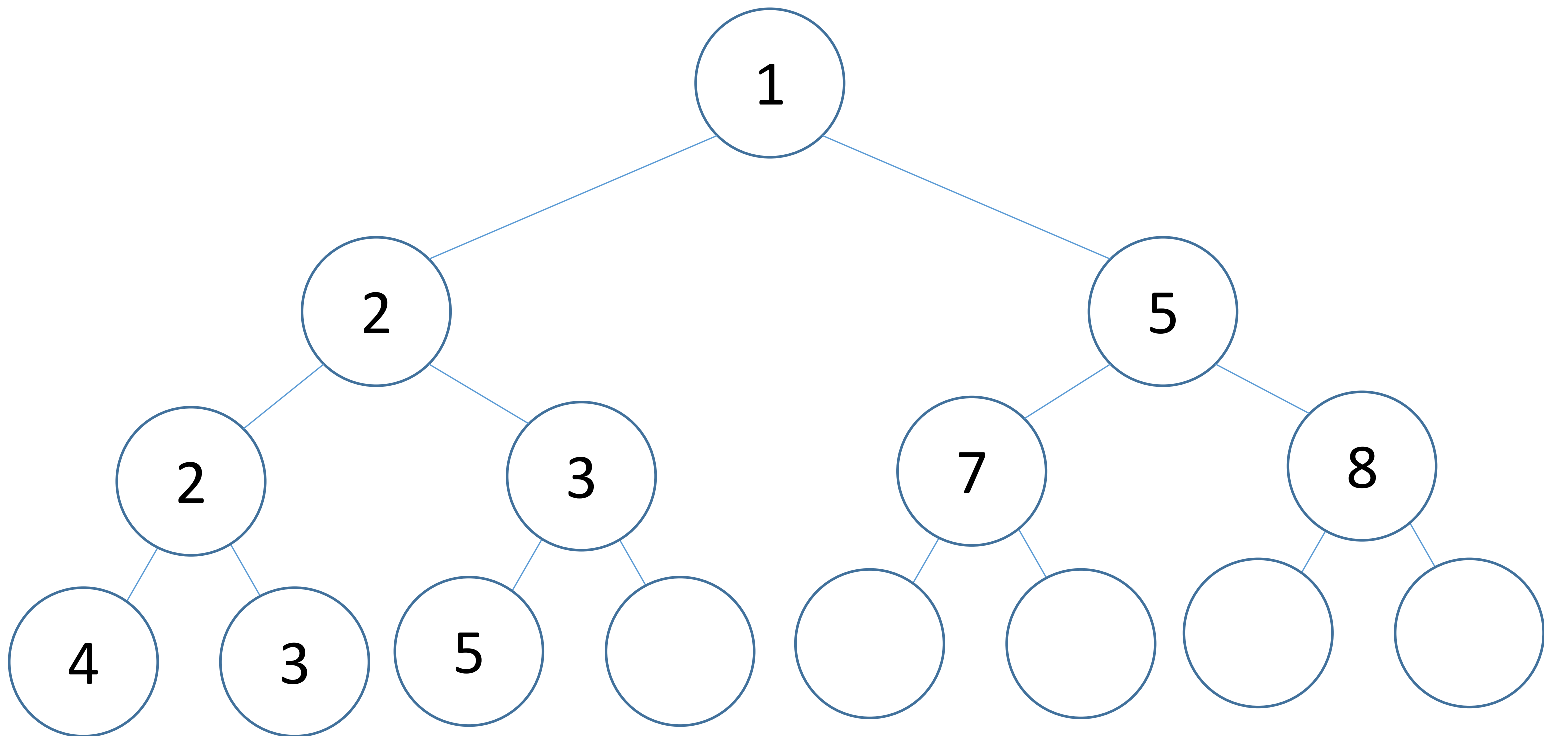
Min Heap

Push 3 // Done



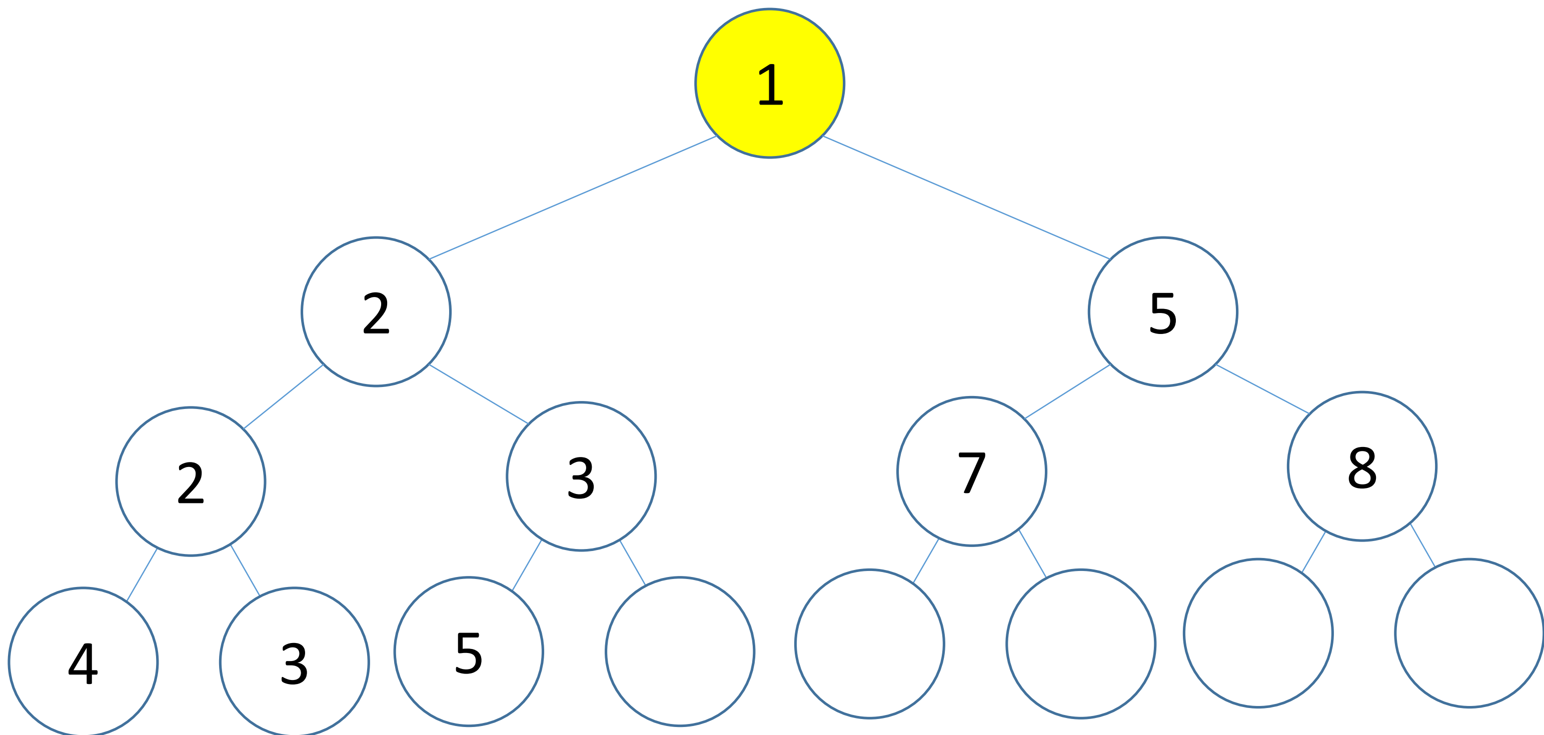
Min Heap

Pop



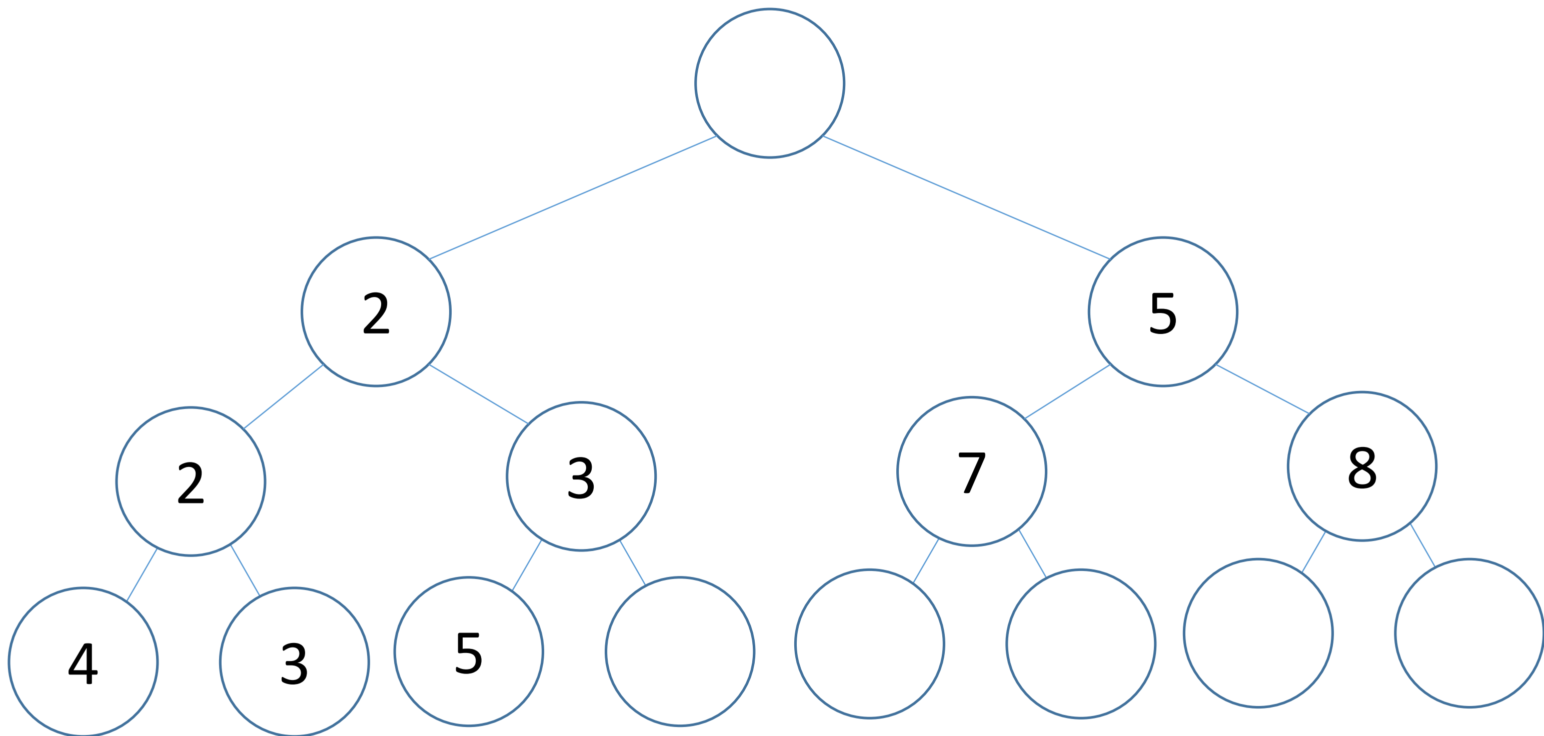
Min Heap

Pop



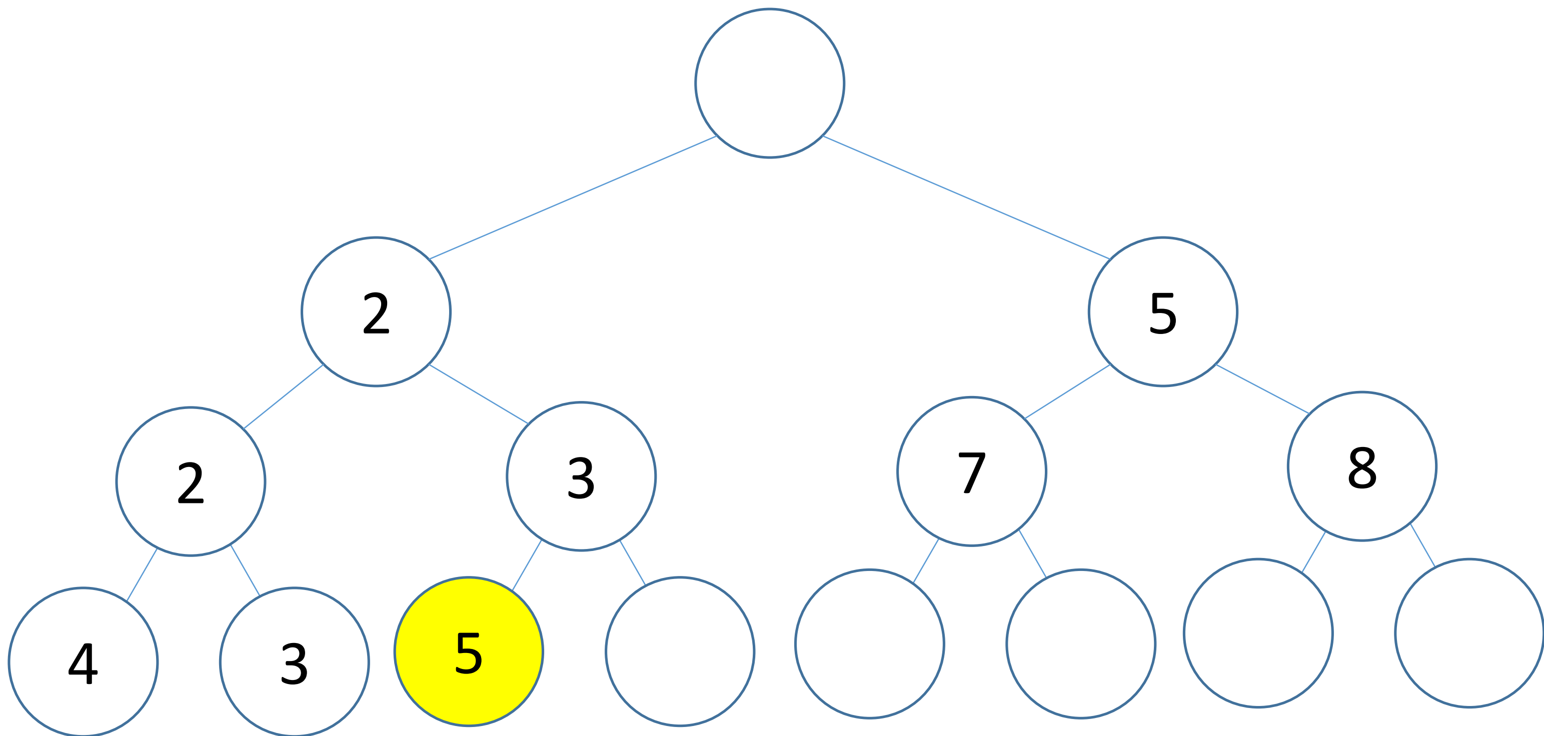
Min Heap

Pop



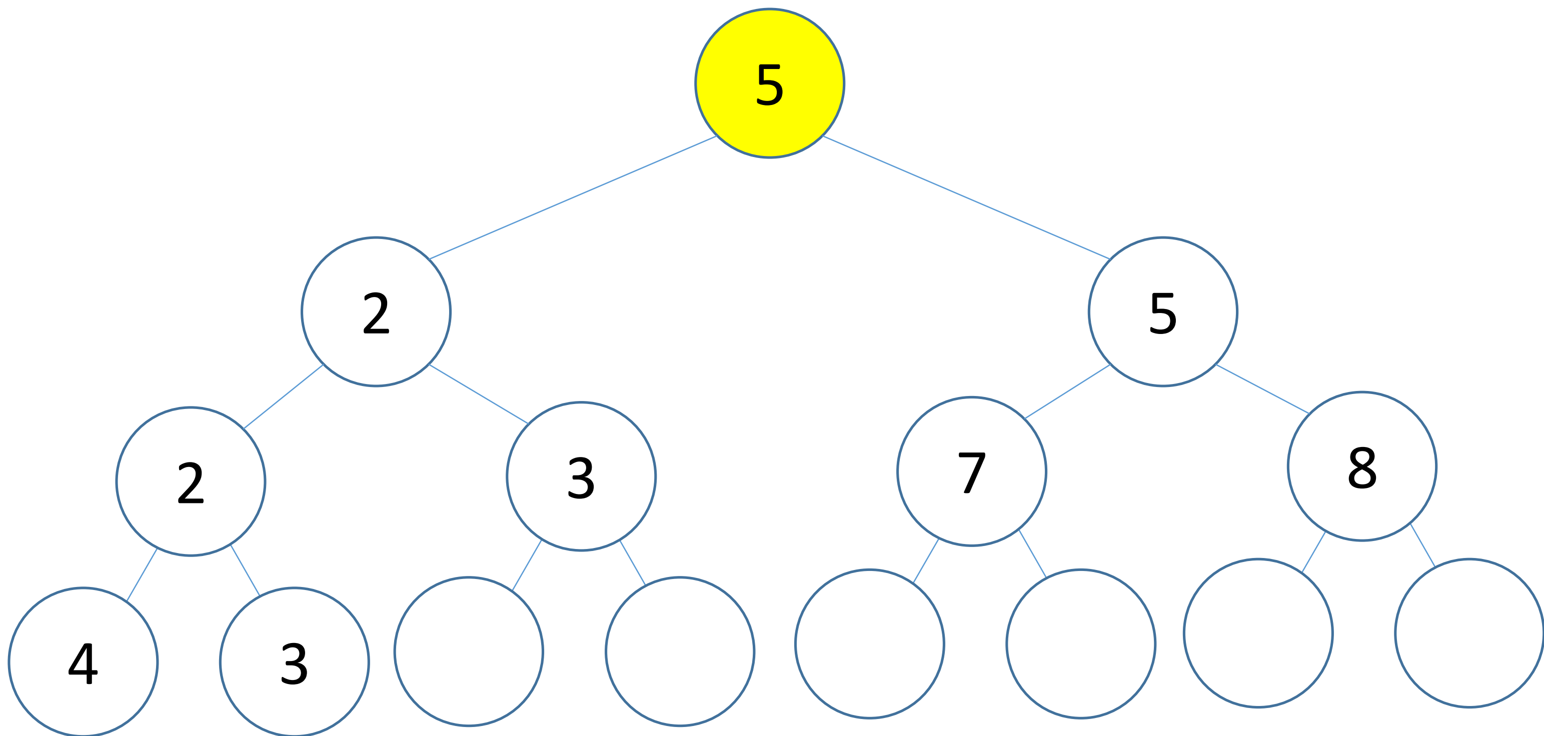
Min Heap

Pop



Min Heap

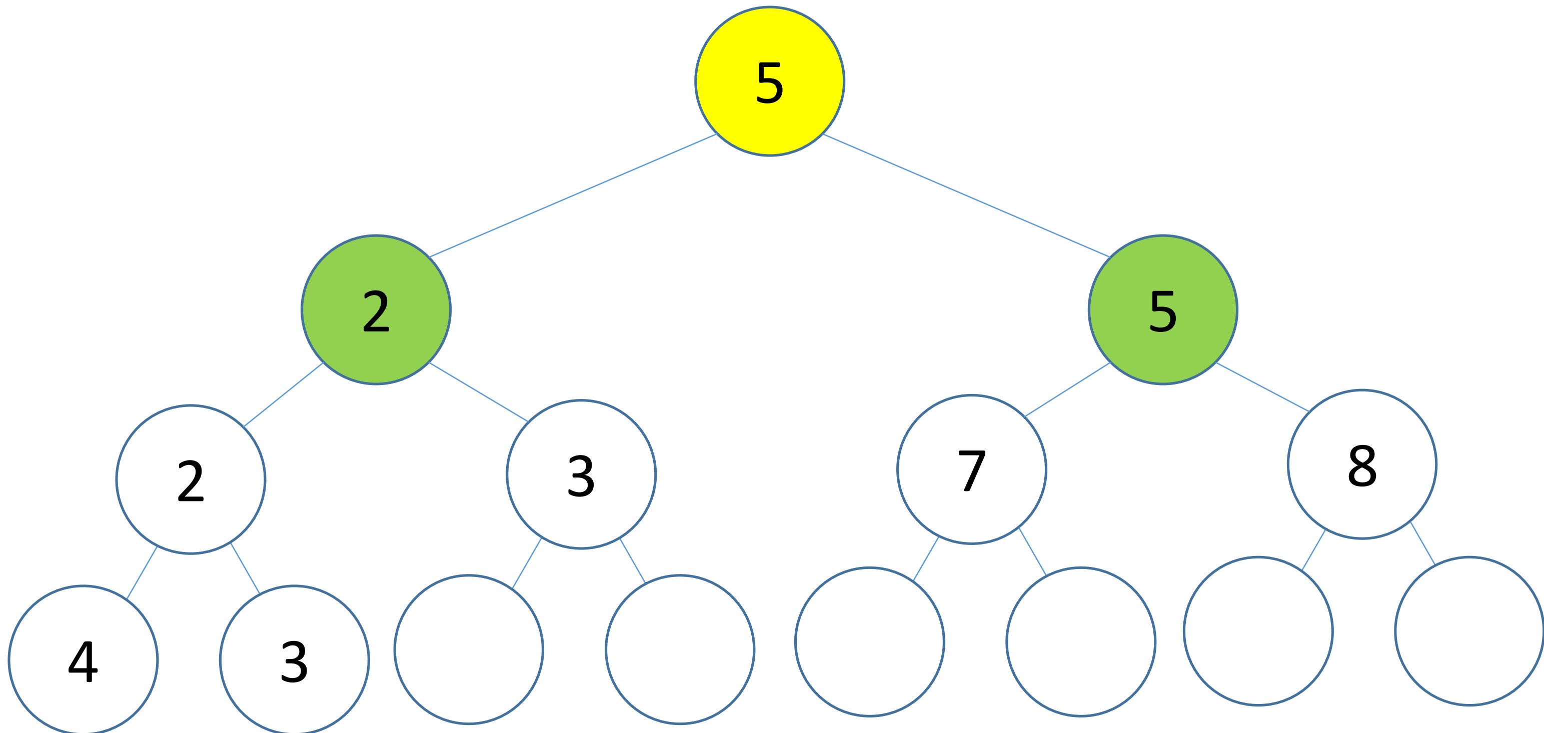
Pop



Min Heap

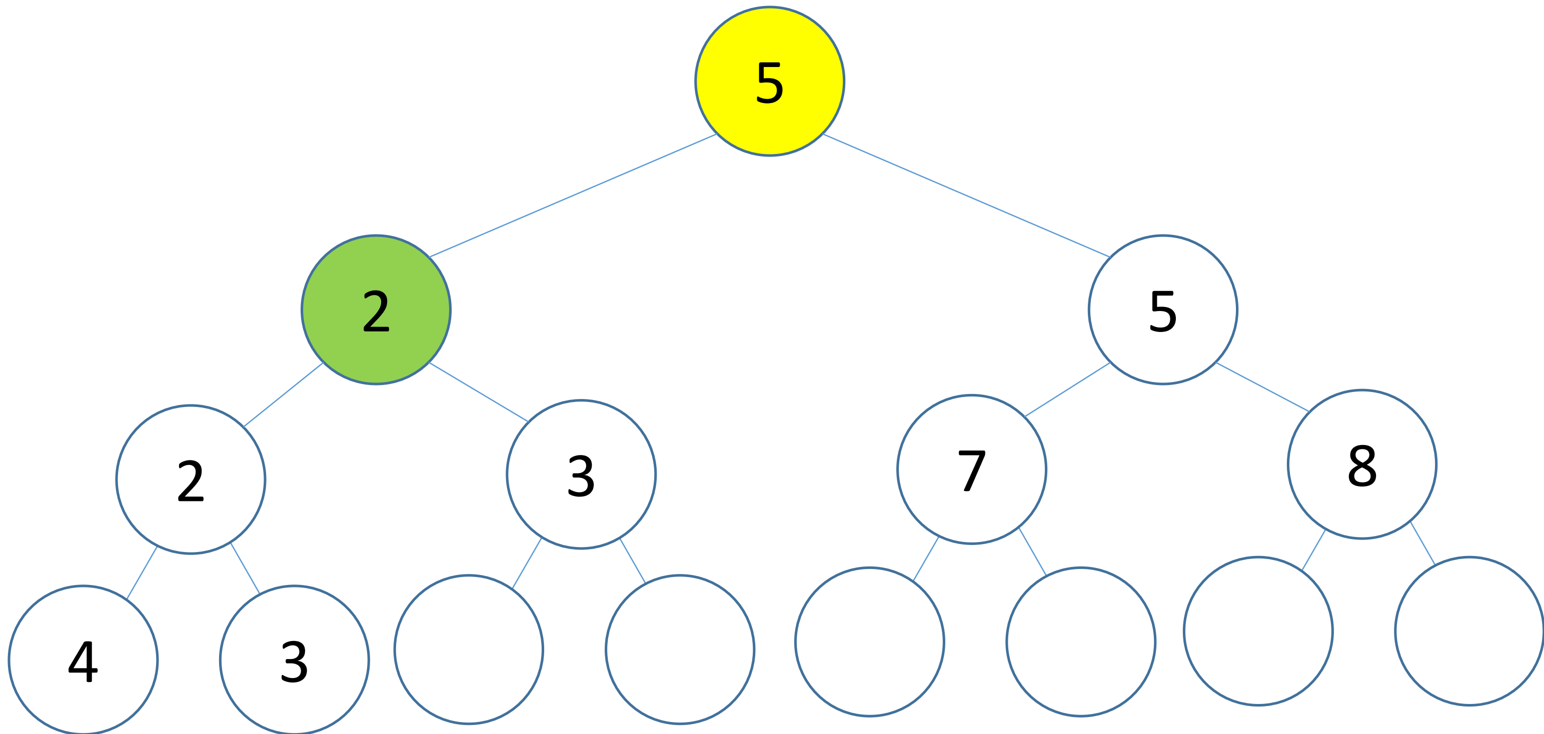
Pop

2 has high priority !



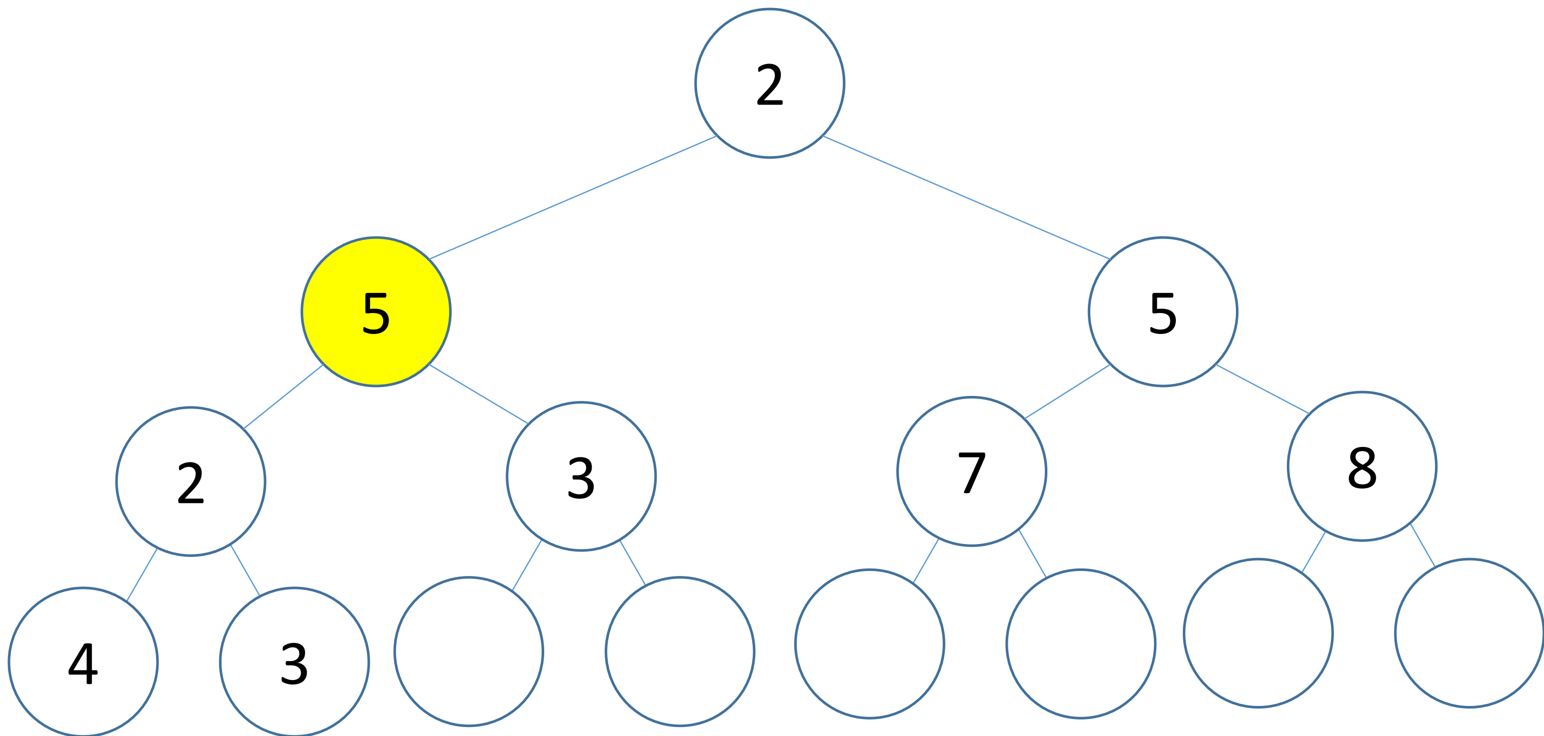
Min Heap

Pop



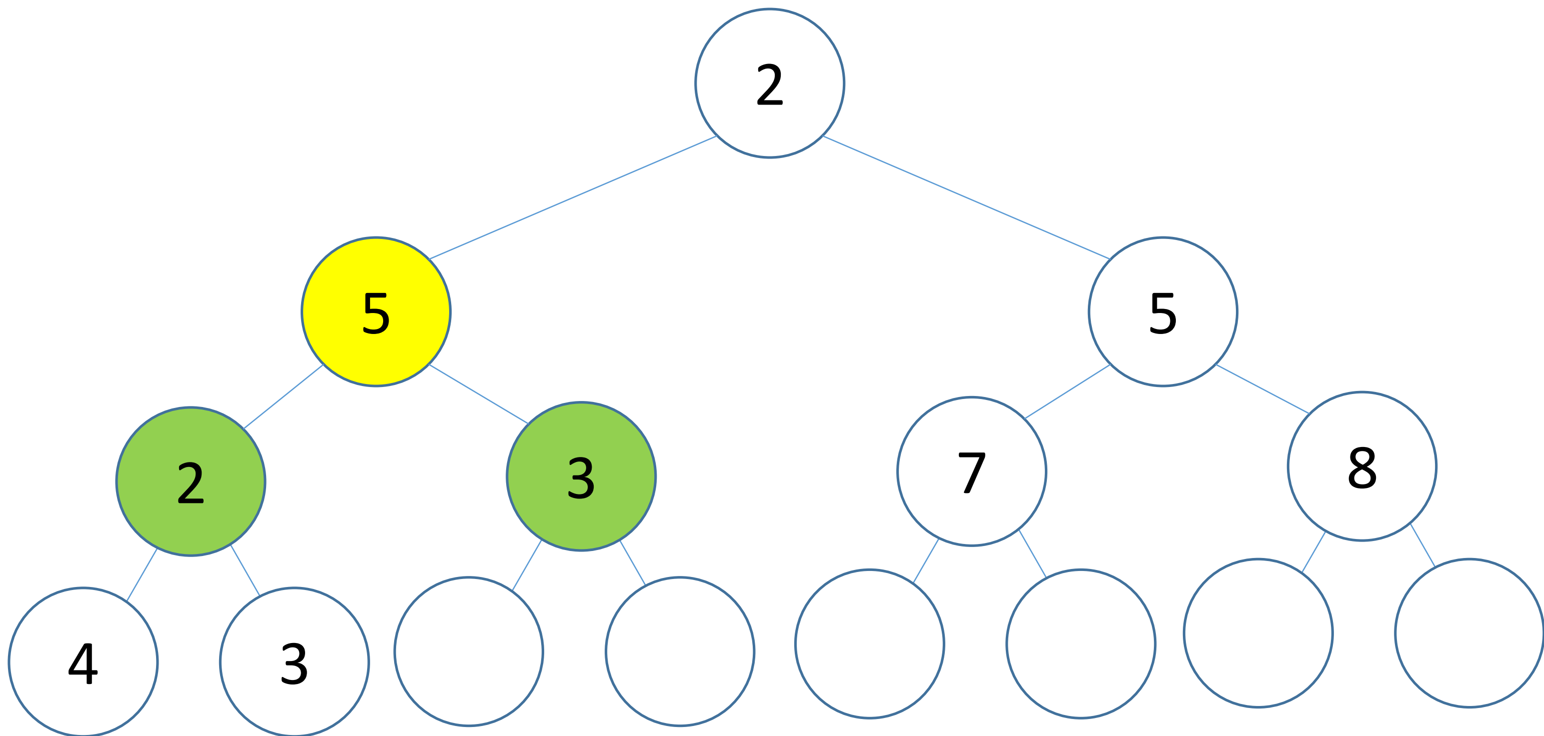
Min Heap

Pop



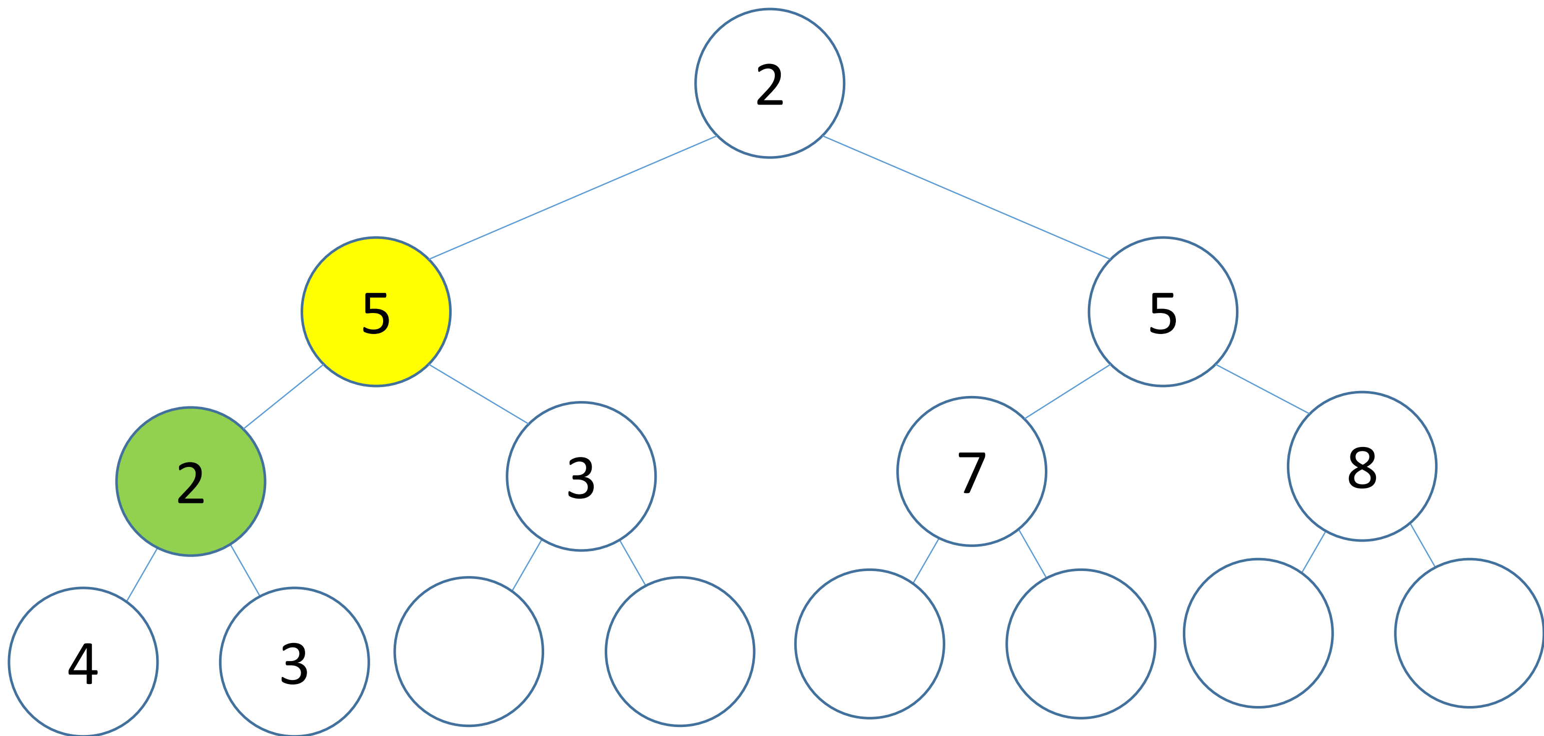
Min Heap

Pop



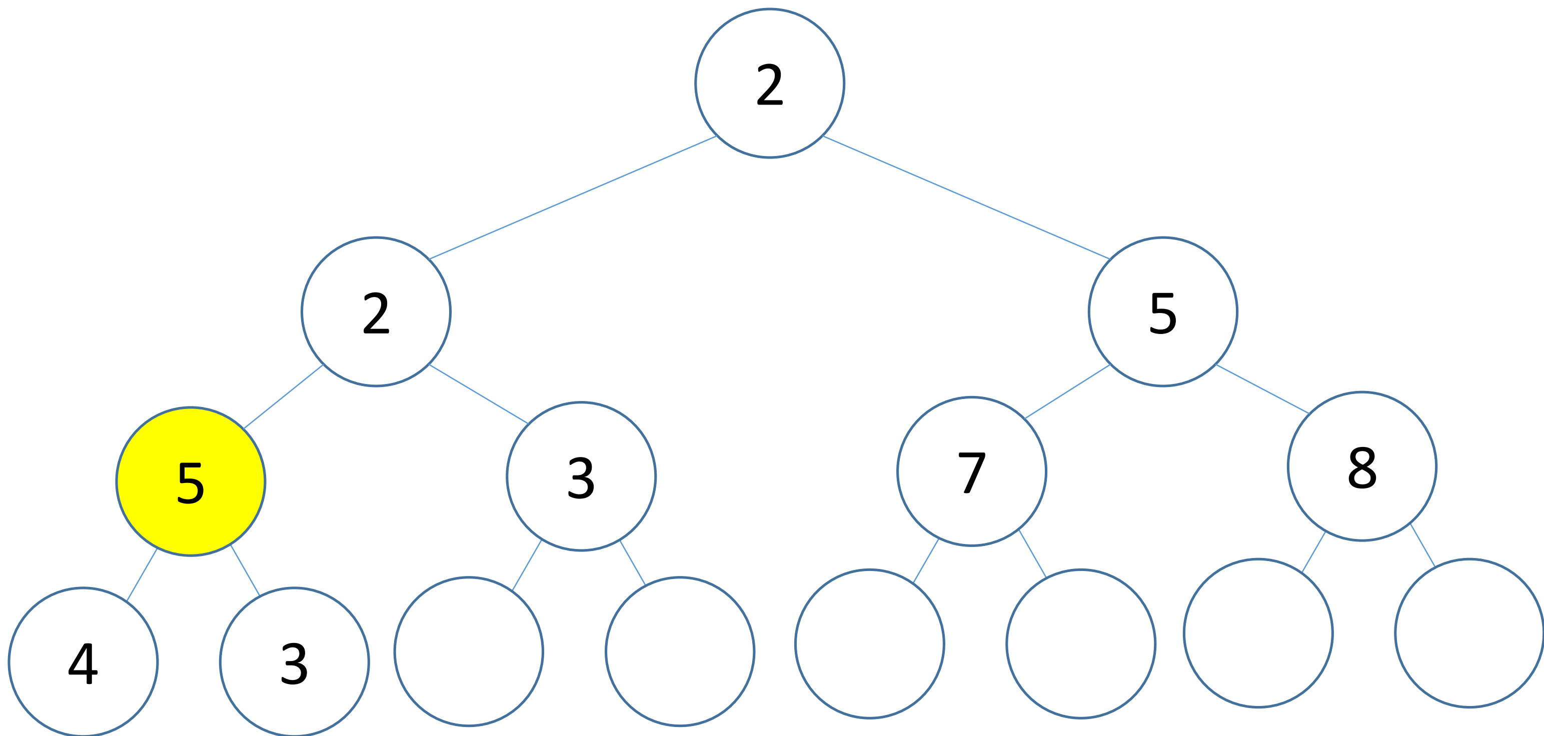
Min Heap

Pop



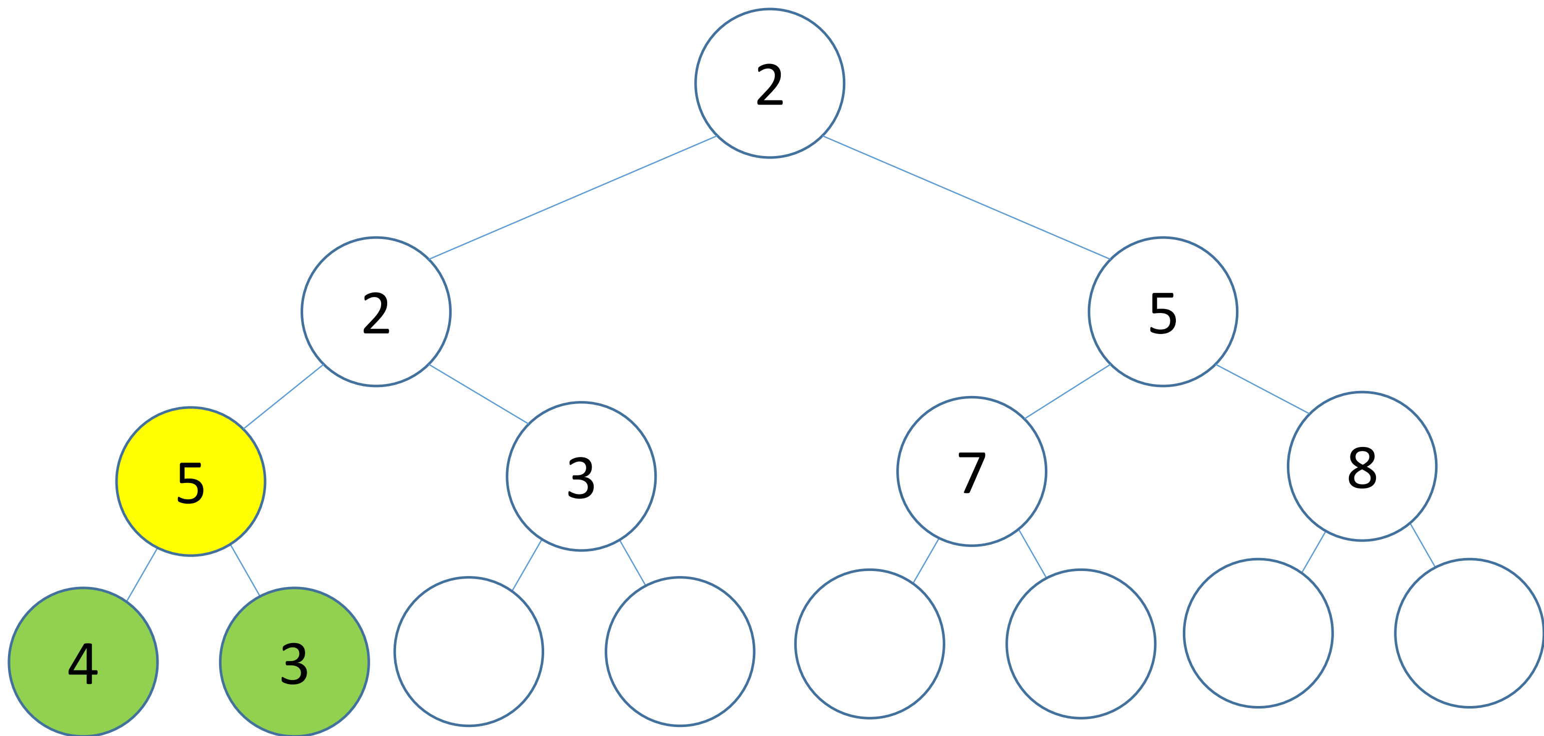
Min Heap

Pop



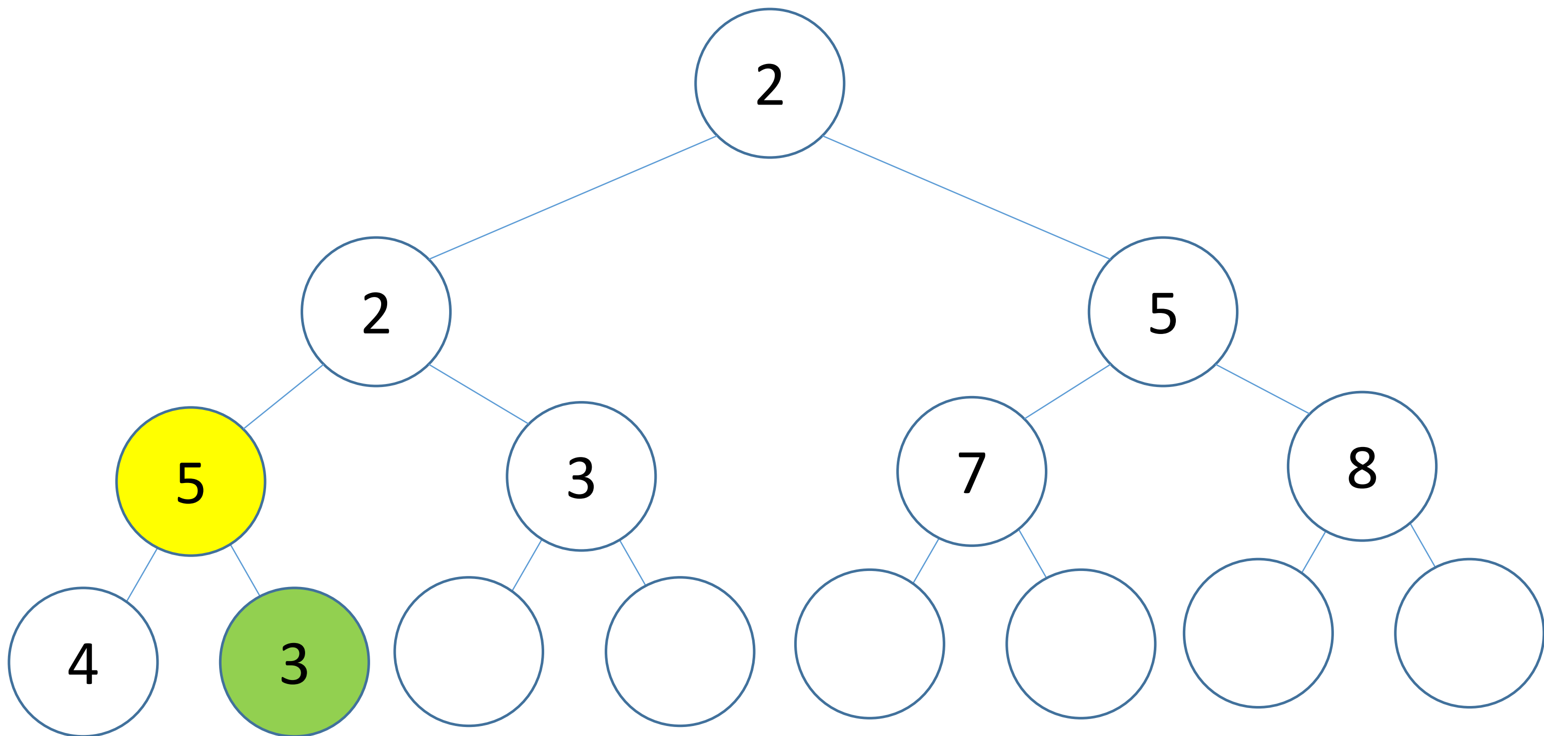
Min Heap

Pop



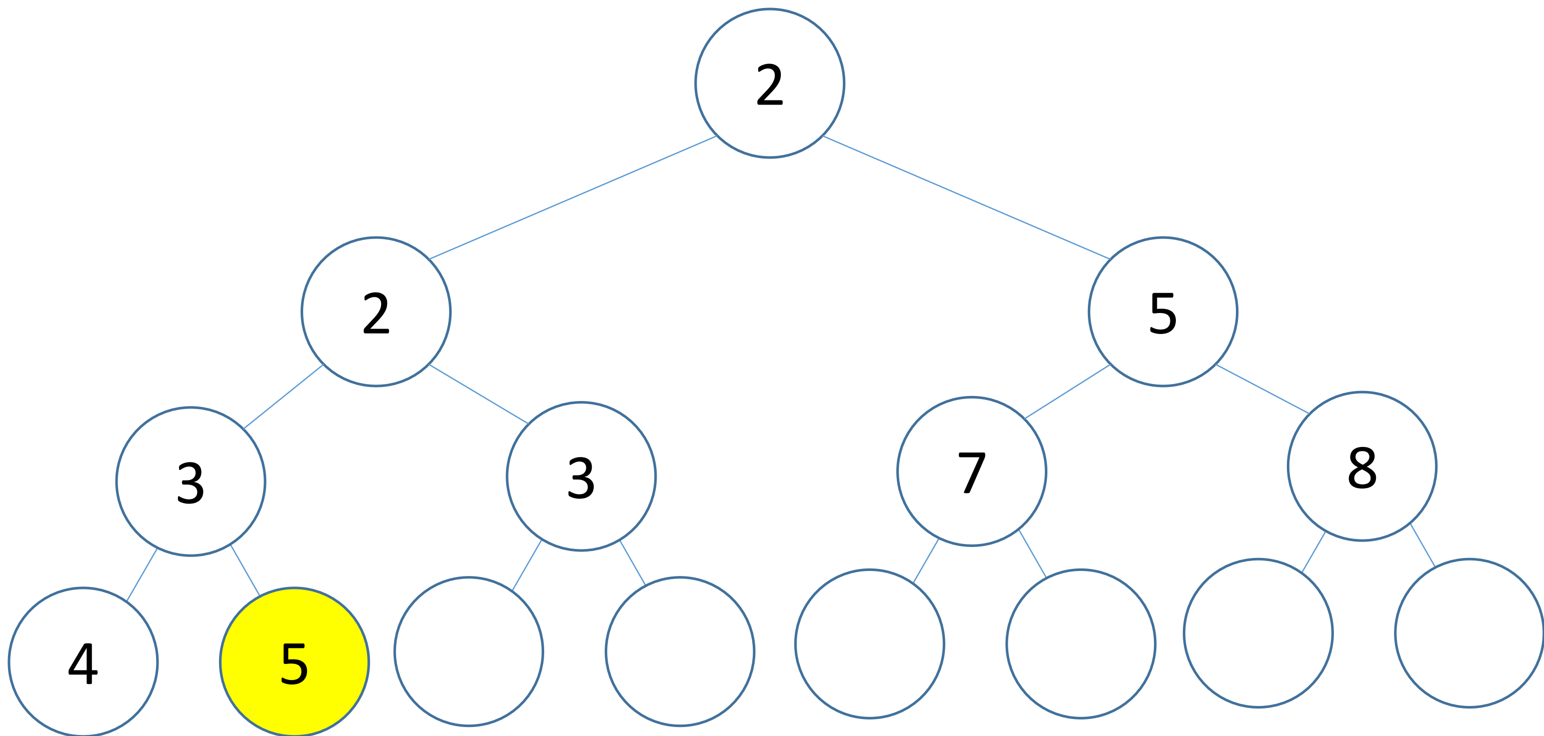
Min Heap

Pop



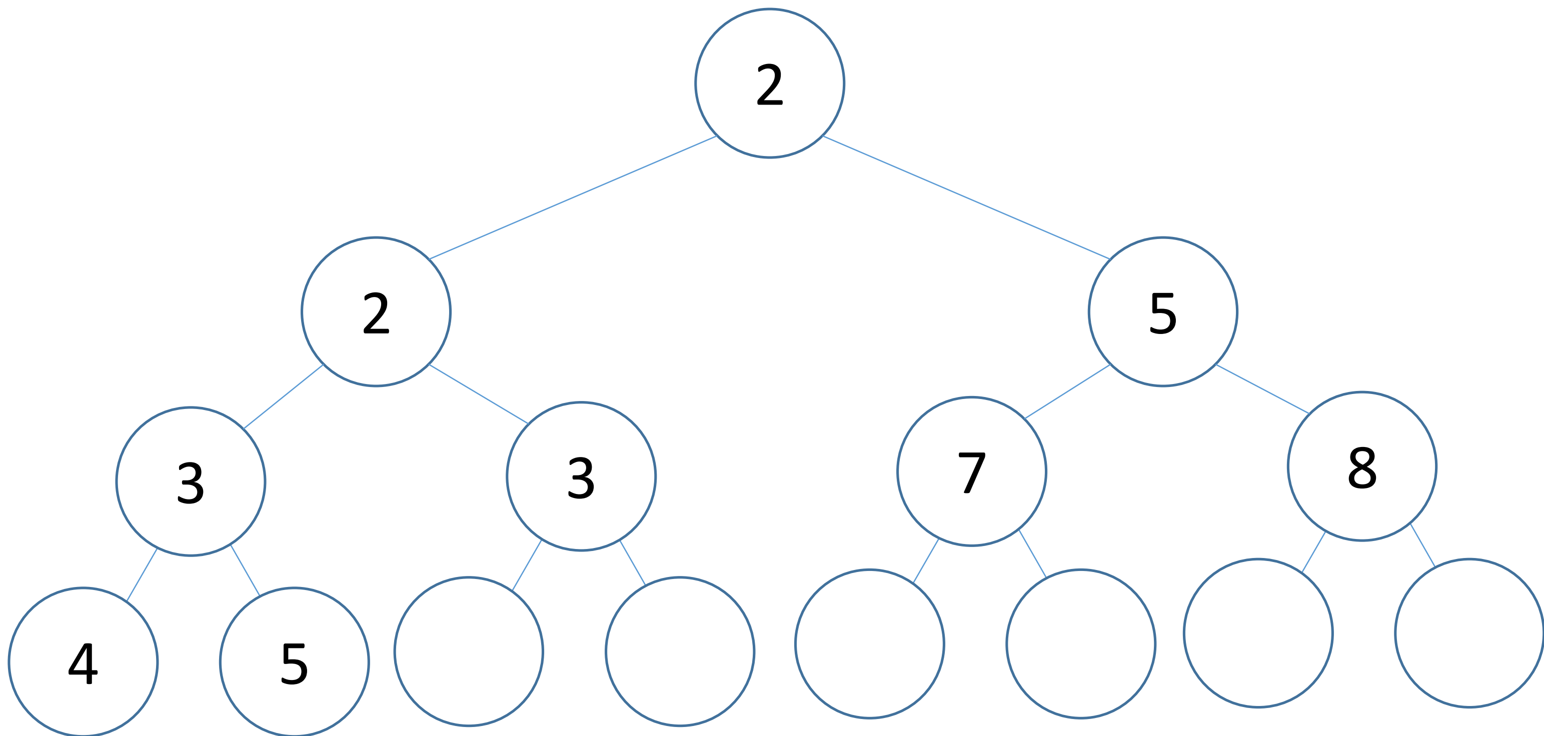
Min Heap

Pop



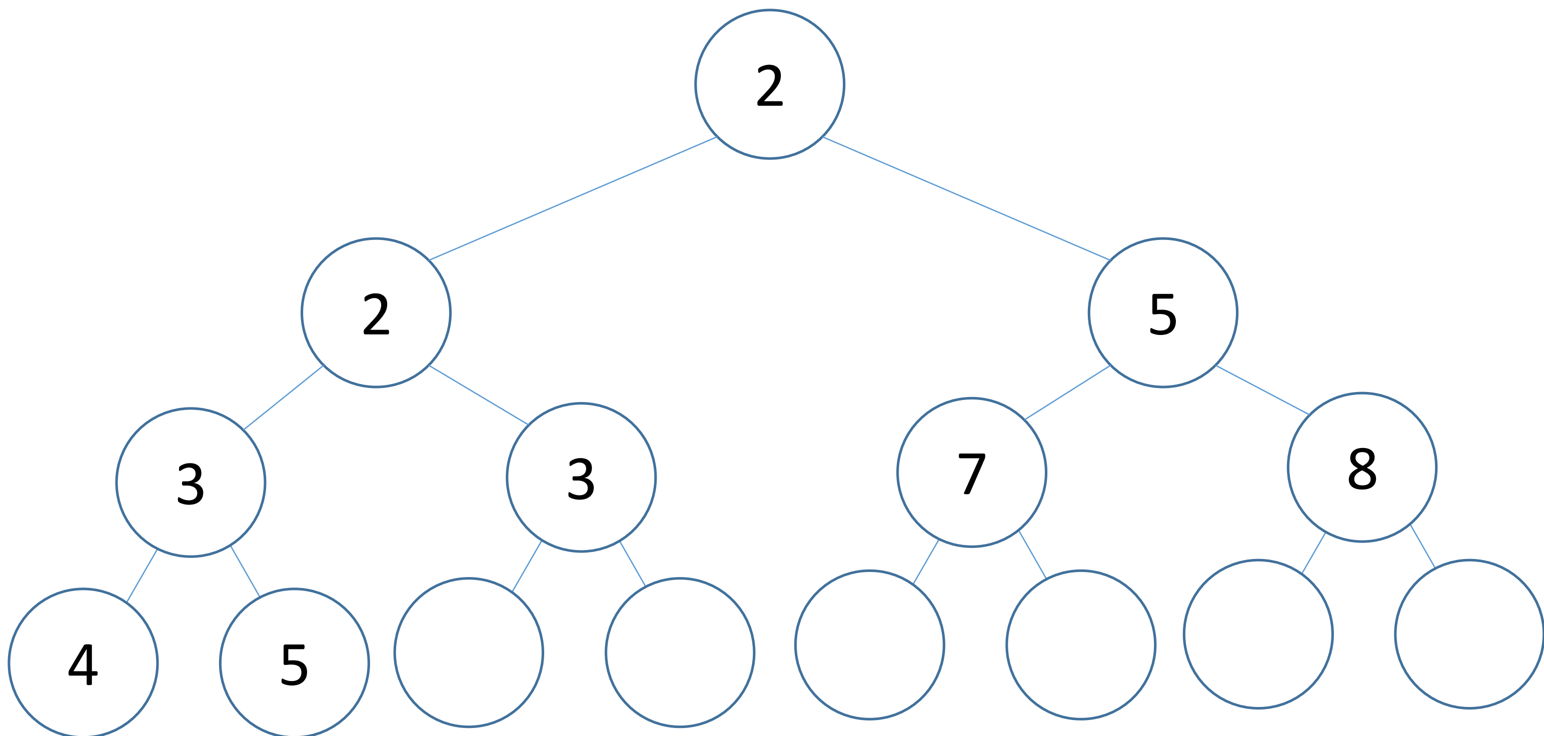
Min Heap

Pop // Done



Min Heap

Pop // Done



Min Heap

Verification : Is this algorithm **true** ?

Efficiency : What time does it **take** ?

Min Heap

Verification : Is this algorithm **true** ? By invarient

Efficiency : What time does it **take** ? **O(log n)** for push & pop

→ Think about height of the tree