Data Structures (II)

Lau Chi Yung

February 24, 2018

Content

Problem Description

N operations:

- `insert` X
- `remove` X
- `exists` X

Constraints:

- $1 \leq N \leq 1000$
- $1 \leq X \leq 1000$

Input

```
7
insert 39
insert 12
insert 74
exists 39
exists 74
remove 12
exists 12
```

Output

```
yes
yes
no
```

Solution: plain array

`insert` 39

39									

`insert` 12

39	12								

...

`insert` 25

39	12	74	93	62	1	50	25		

Solution: plain array

`insert 39`

39									

`insert 12`

39	12								

...

`insert 25`

39	12	74	93	62	1	50	25		

`remove 12`

39		74	93	62	1	50	25			or
39	74	93	62	1	50	25				or
39	25	74	93	62	1	50				?

Solution: plain array

`insert` 39	39										
`insert` 12	39	12									
				…							
`insert` 25	39	12	74	93	62	1	50	25			
	39		74	93	62	1	50	25			or
`remove` 12	39	74	93	62	1	50	25				or
	39	25	74	93	62	1	50				?

`exists` 74 perform linear search

Solution: plain array

N operations:

- `insert` X
- `remove` X
- `exists` X

Constraints:

- $1 \leq N \leq 1000$
- $1 \leq X \leq 1000$

Time complexity:

`insert`	$O(1)$
`remove`	$O(N)$
`exists`	$O(N)$

For N operations, worst case $O(N^2)$

Solution: sorted array

`insert` 39	39									
`insert` 12	12	39								

...

`insert` 25	1	12	25	39	50	62	74	93		

Solution: sorted array

`insert 39`	39									
`insert 12`	12	39								
			...							
`insert 25`	1	12	25	39	50	62	74	93		
`remove 12`	1	25	39	50	62	74	93			

Solution: sorted array

`insert 39`	39									
`insert 12`	12	39								
			...							
`insert 25`	1	12	25	39	50	62	74	93		
`remove 12`	1	25	39	50	62	74	93			

`exists 74` perform binary search

Solution: sorted array

N operations:

- `insert` X
- `remove` X
- `exists` X

Constraints:

- $1 \leq N \leq 1000$
- $1 \leq X \leq 1000$

Time complexity:

`insert`	$O(N)$
`remove`	$O(N)$
`exists`	$O(\log N)$

For N operations, worst case $O(N^2)$

Solution: hybrid

N operations:

- `insert` X
- `remove` X
- `exists` X

Constraints:

- $1 \leq N \leq 1000$
- $1 \leq X \leq 1000$

Dominant operation	Solution array
`insert`	plain or sorted?
`exists`	plain or sorted?

Solution: hybrid

N operations:

- `insert` X
- `remove` X
- `exists` X

Constraints:

- $1 \leq N \leq 1000$
- $1 \leq X \leq 1000$

Dominant operation	Solution array
`insert`	plain or sorted?
`exists`	plain or sorted?

Solution: hybrid

N operations:

- `insert` X
- `remove` X
- `exists` X

Constraints:

- $1 \leq N \leq 1000$
- $1 \leq X \leq 1000$

Dominant operation	Solution array
`insert`	plain or sorted?
`exists`	plain or sorted?

Solution: hybrid

N operations:

- `insert` X
- `remove` X
- `exists` X

Constraints:

- $1 \leq N \leq 1000$
- $1 \leq X \leq 1000$

Dominant operation	Solution array
`insert`	plain or sorted?
`exists`	plain or sorted?

For N operations, still $O(N^2)$

Problem Description - Level Up

N operations:

- `insert` X
- `remove` X
- `exists` X

Constraints:

- $1 \leq N \leq 10000$
- $1 \leq X \leq 1000$

Solution: counting array

Similar idea with counting sort

`insert` 12

											1					
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

`insert` 17

											1					1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

`insert` 6

					1						1					1
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17

Solution: counting array

Similar idea with counting sort

`insert` 12

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
											1					

`insert` 17

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
											1					1

`insert` 6

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
					1						1					1

`remove` 12

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
					1											1

Solution: counting array

Similar idea with counting sort

`insert` 12

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
											1					

`insert` 17

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
											1					1

`insert` 6

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
					1						1					1

`remove` 12

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
					1											1

`exists` 12 array lookup

Solution: counting array

N operations:

- `insert` X
- `remove` X
- `exists` X

Constraints:

- $1 \leq N \leq 10000$
- $1 \leq X \leq 1000$

Time complexity:

`insert`	$O(1)$
`remove`	$O(1)$
`exists`	$O(1)$

For N operations, worst case $O(N)$

Problem Description - Level Up

N operations:

- `insert` X
- `remove` X
- `exists` X

Constraints:

- $1 \leq N \leq 10000$
- $1 \leq X \leq 10^8$
- Memory limit: 32MB

Problem Description - Level Up

N operations:

- `insert` X
- `remove` X
- `exists` X

Constraints:

- $1 \leq N \leq 10000$
- $1 \leq X \leq 10^8$
- Memory limit: 32MB

Can we use counting array?

```
int a[100000000 + 1];
```

$4 \times 10^8 = 400\text{MB}$ MLE

Problem Description - Level Up

N operations:

- `insert` X
- `remove` X
- `exists` X

Constraints:

- $1 \leq N \leq 10000$
- $1 \leq X \leq 10^8$
- Memory limit: 32MB

"`int`" contains 32 bits

```
int a[3125000 + 1];
```

$4 \times 3125000 = 12.5\text{MB}$

`insert` X	`a[X/32] |= 1<<X%32`
`remove` X	`a[X/32] &= ~(1<<X%32)`
`exists` X	`a[X/32] & 1<<X%32`

Problem Description - Level Up Again

N operations:

- `insert` X
- `remove` X
- `exists` X

Constraints:

- $1 \leq N \leq 10000$
- $1 \leq X \leq 10^{18}$
- Memory limit: 32MB

Problem Description - Level Up Again

N operations:

- `insert` X
- `remove` X
- `exists` X

Constraints:

- $1 \leq N \leq 10000$
- $1 \leq X \leq 10^{18}$
- Memory limit: 32MB

Observation:
There can be at most 10000 distinct values of X

Problem Description - Level Up Again

N operations:

- `insert` X
- `remove` X
- `exists` X

Constraints:

- $1 \leq N \leq 10000$
- $1 \leq X \leq 10^{18}$
- Memory limit: 32MB

Observation:
There can be at most 10000 distinct values of X

Perform *discretization*:

1. Preprocess all X into a sorted array
2. Using binary search, transform the range of X from $[1, 10^{18}]$ to $[1, 10000]$
3. Then use counting array

Problem Description - Level Up Yet Again

N operations:

- `insert` X
- `remove` X
- `exists` X

Constraints:

- $1 \leq N \leq 10000$
- $1 \leq X \leq 10^{18}$
- Memory limit: 32MB
- Preprocess of operations is not allowed (how?)

Problem Description - Level Up Yet Again

N operations:

- `insert` X
- `remove` X
- `exists` X

Constraints:

- $1 \leq N \leq 10000$
- $1 \leq X \leq 10^{18}$
- Memory limit: 32MB
- Preprocess of operations is not allowed (how?)

- Perhaps the task is interactive

Problem Description - Level Up Yet Again

N operations:

- `insert` X
- `remove` X
- `exists` X

Constraints:

- $1 \leq N \leq 10000$
- $1 \leq X \leq 10^{18}$
- Memory limit: 32MB
- Preprocess of operations is not allowed (how?)

- Perhaps the task is interactive
- Perhaps each operation's input depends on the output of previous operations

Observation

- We will encounter only 10000 integers in $[1, 10^{18}]$
- Intuitively, X mod 10^8 will *probably* be distinct
- Assume for any $X_i \neq X_j$, $X_i \bmod 10^8 \neq X_j \bmod 10^8$
- We can replace all X with X mod 10^8 and use bitwise counting array!
- How *likely* will our assumption hold?

Birthday problem

Probability that all 23 people have distinct birthdays

$$P = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \cdots \times \frac{343}{365} \approx 0.492703$$

For 70 people, the probablity is 0.000840

Birthday problem

Probability that all 23 people have distinct birthdays

$$P = \frac{365}{365} \times \frac{364}{365} \times \frac{363}{365} \times \cdots \times \frac{343}{365} \approx 0.492703$$

For 70 people, the probablity is 0.000840

Assume X is uniformly distributed over $[1, 10^{18}]$

Then X mod 10^8 is uniformly distributed over $[0, 10^8 - 1]$

Probability that all 10000 X mod 10^8 are distinct

$$P = \frac{10^8}{10^8} \times \cdots \times \frac{10^8 - 9999}{10^8} \approx 0.606551$$

Not too bad.

Birthday problem

$$P = \frac{10^8}{10^8} \times \cdots \times \frac{10^8 - 9999}{10^8} \approx 0.606551$$

Had we used 2×10^8 (which amounts to 25MB in bitwise counting array),

$$P = \frac{2 \times 10^8}{2 \times 10^8} \times \cdots \times \frac{2 \times 10^8 - 9999}{2 \times 10^8} \approx 0.778817$$

Intuitively, with more buckets available, less likely will collision occur.

This does not work well for batched testcases.

Probablility that we get all 5 testcases correct in a batch $= 0.778817^5 = 0.286534$

Collision Handling 1

Rather than giving up on collisions, we try to handle them.

Want to map X into X mod 10.

`insert` X: put X in the *nearest* empty bucket after X mod 10

	0	1	2	3	4	5	6	7	8	9
`insert` 12			12							
`insert` 34			12		34					
`insert` 22			12	22	34					
`insert` 42			12	22	34	42				

Collision Handling 1

`remove` X: search for the bucket containing X, replace it with -1

		12	22	34	42				
0	1	2	3	4	5	6	7	8	9

`remove` 22

		12	-1	34	42				
0	1	2	3	4	5	6	7	8	9

`exists` X: scan from bucket X mod 10 until empty bucket

Collision Handling 1

`insert` X $O(1)$ average

`remove` X $O(1)$ average

`exists` X $O(1)$ average

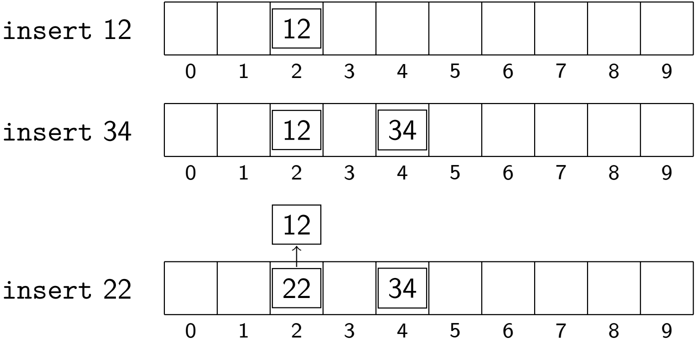
This kind of collision handling is called *Open Addressing*

Variants

- Linear probing
- Quadratic probing

Collision Handling 2

Store all collisions in linked lists.



Collision Handling 2

`insert` X	add item to the linked list at bucket X mod 10 $O(1)$ average
`remove` X	remove item in the linked list at bucket X mod 10 $O(1)$ average
`exists` X	scan the linked list at bucket X mod 10 $O(1)$ average

This kind of collision handling is called *Separate Chaining*

Hash Table

The data structure that we built is called *Hash Table*.

The core function that we used

$$h(X) = X \bmod P$$

is called the *hash function*, which maps *something* into a range of integers that we can handle.

Hash Table

The data structure that we built is called *Hash Table*.

The core function that we used

$$h(X) = X \bmod P$$

is called the *hash function*, which maps *something* into a range of integers that we can handle.

Is it better to use a prime number for P?

Hash Table

The performance of a hash table is determined by the load factor $\frac{N}{K}$, where N is the number of items and K is the number of buckets.

For *Separate Chaining*, length of linked lists is proportional to load factor

Load factor	Wasted space	Performance
smaller	smaller or larger?	faster or slower?
larger	smaller or larger?	faster or slower?

Hash Table

The performance of a hash table is determined by the load factor $\frac{N}{K}$, where N is the number of items and K is the number of buckets.

For *Separate Chaining*, length of linked lists is proportional to load factor

Load factor	Wasted space	Performance
smaller	smaller or larger?	faster or slower?
larger	smaller or larger?	faster or slower?

Hash Table

The performance of a hash table is determined by the load factor $\frac{N}{K}$, where N is the number of items and K is the number of buckets.

For *Separate Chaining*, length of linked lists is proportional to load factor

Load factor	Wasted space	Performance
smaller	smaller or larger?	faster or slower?
larger	smaller or larger?	faster or slower?

Hash Table

The performance of a hash table is determined by the load factor $\frac{N}{K}$, where N is the number of items and K is the number of buckets.

For *Separate Chaining*, length of linked lists is proportional to load factor

Load factor	Wasted space	Performance
smaller	smaller or larger?	faster or slower?
larger	smaller or larger?	faster or slower?

C++ Standard Template Library

- C++11 only

```cpp
#include <unordered_set>
using namespace std;
...
unordered_set<int> s;
s.insert(123);
int x = s.count(123); //1
s.erase(123);
```

```cpp
#include <unordered_map>
using namespace std;
...
unordered_map<int, int> m;
m[123] = 456;
int x = m[123]; //456
m.erase(123);
```

Problem Description

N operations:

- `insert` X
- `get min`
  (print the minimum item)
- `remove min`
  (remove the minimum item)

Constraints:

- $1 \leq N \leq 10000$
- $1 \leq X \leq 10^{18}$

Input

```
6
insert 12
insert 74
get min
insert 39
remove min
get min
```

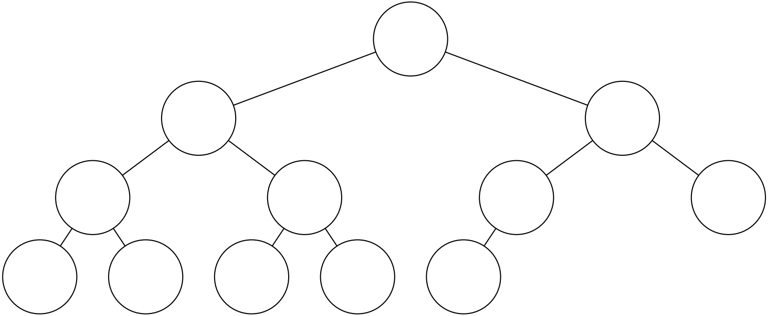
Output

```
12
39
```

Binary Heap

Shape property: compelete binary tree

- The bottom level is filled from left to right
- All above levels are fully filled

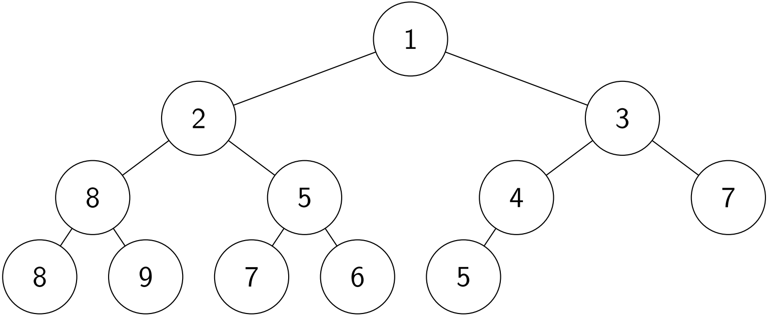


Binary Heap

Shape property: compelete binary tree

- The bottom level is filled from left to right
- All above levels are fully filled

Heap property: every node $\geq$ its parent



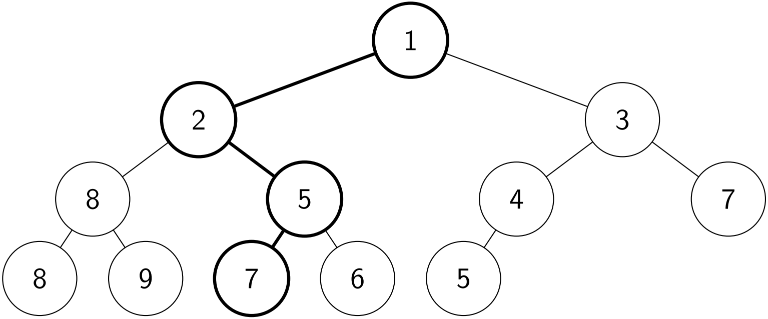
Binary Heap

Shape property: compelete binary tree

- The bottom level is filled from left to right
- All above levels are fully filled

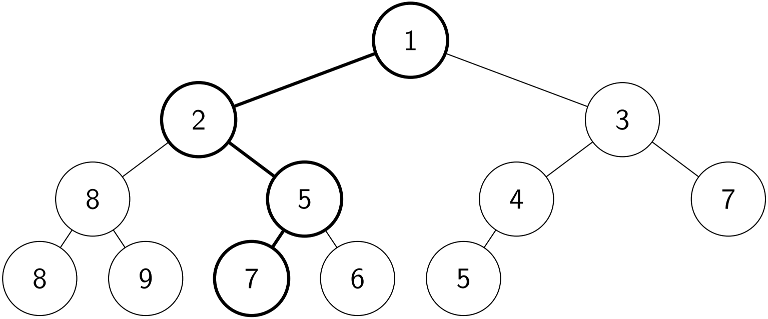
Heap property: every node $\geq$ its parent

Are all root-to-leaf paths sorted?



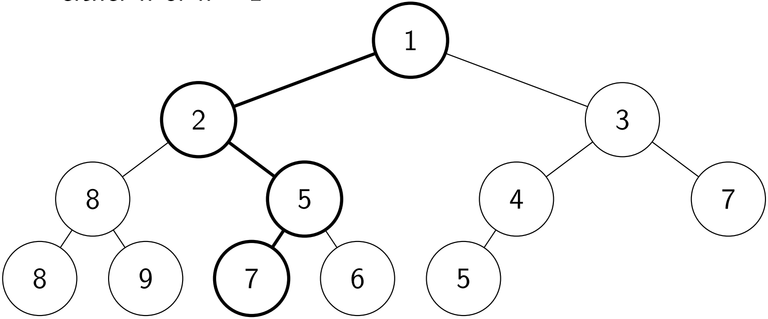
Binary Heap

- Number of nodes $= N$
- Tree height $h =$ maximum number of edges from root to leaf
- $2^h \leq N \leq 2^{h+1} - 1$
- $h \leq \log_2 N = O(\log N)$
- What are the possible lengths of all root-to-leaf paths?



Binary Heap

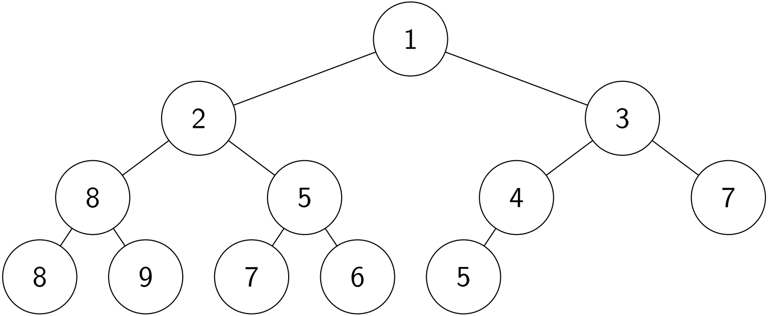
- Number of nodes $= N$
- Tree height $h =$ maximum number of edges from root to leaf
- $2^h \leq N \leq 2^{h+1} - 1$
- $h \leq \log_2 N = O(\log N)$
- What are the possible lengths of all root-to-leaf paths?
- either h or $h - 1$



Binary Heap

`insert` 2

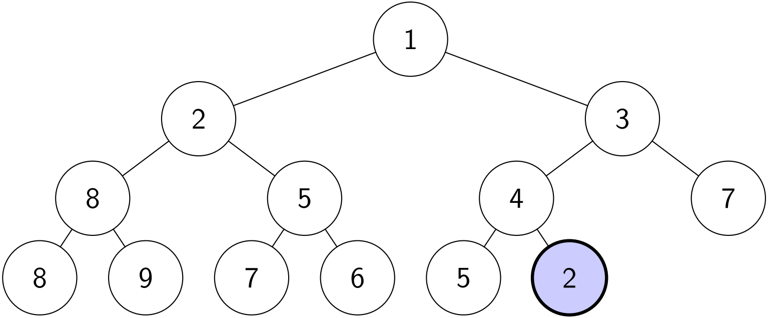
- Add new node at the bottom



Binary Heap

`insert` 2

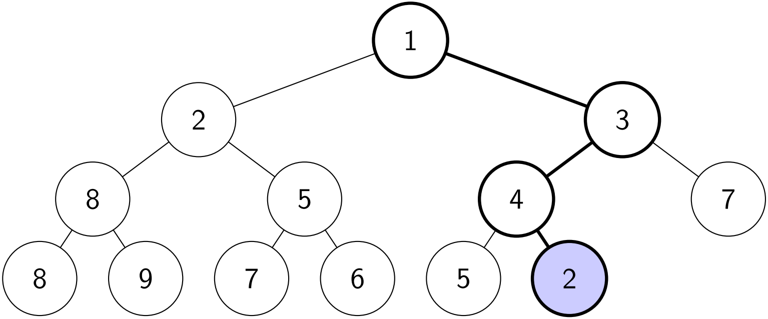
- Add new node at the bottom
- Heap property violated
- How many root-to-leaf paths are not sorted?



Binary Heap

`insert` 2

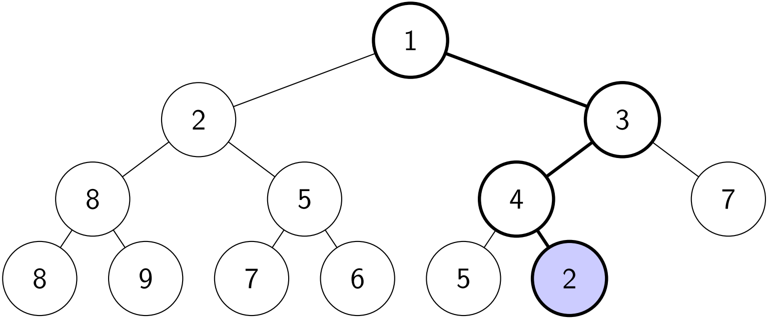
- Add new node at the bottom
- Heap property violated
- How many root-to-leaf paths are not sorted?
- Only one, let's (merge/quick/bubble/insertion/selection) sort that path



Binary Heap

`insert` 2

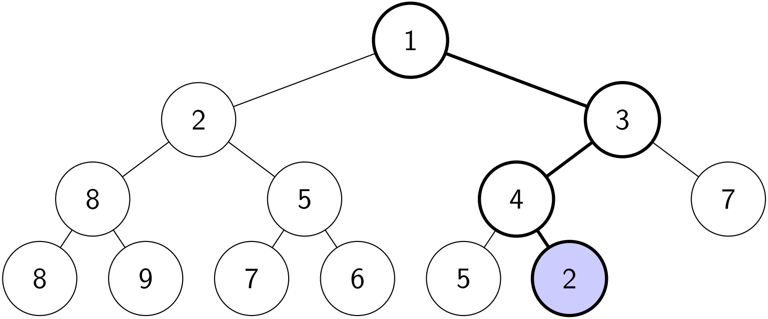
- Add new node at the bottom
- Heap property violated
- How many root-to-leaf paths are not sorted?
- Only one, let's insertion sort that path



Binary Heap

`insert` 2

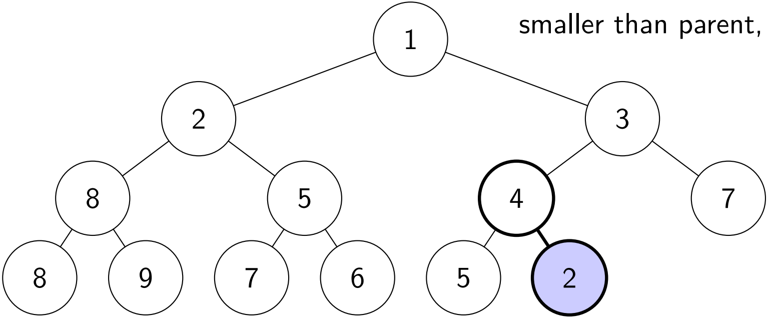
- Add new node at the bottom
- Heap property violated
- How many root-to-leaf paths are not sorted?
- We call it *sift-up*



Binary Heap

`insert` 2

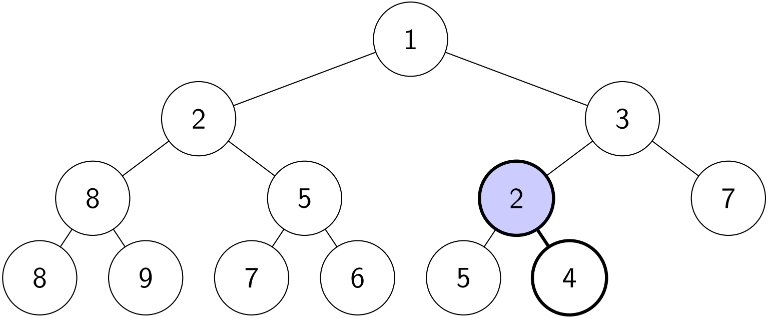
- Add new node at the bottom
- Heap property violated
- How many root-to-leaf paths are not sorted?
- We call it *sift-up*



Binary Heap

`insert` 2

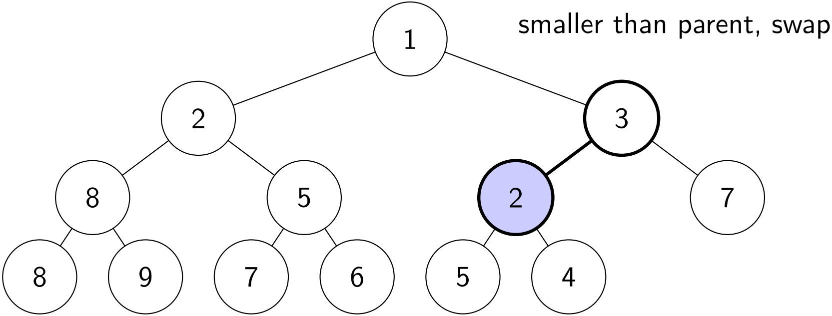
- Add new node at the bottom
- Heap property violated
- How many root-to-leaf paths are not sorted?
- We call it *sift-up*



Binary Heap

`insert` 2

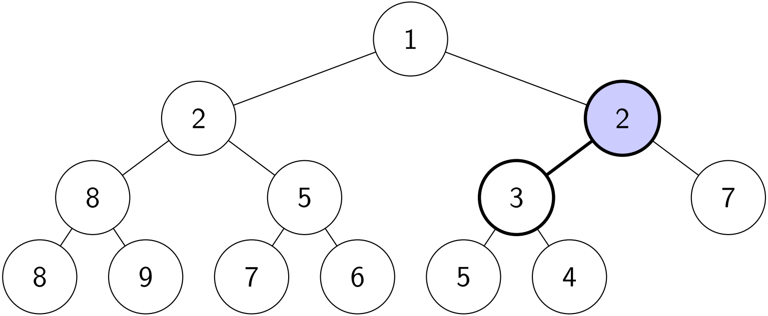
- Add new node at the bottom
- Heap property violated
- How many root-to-leaf paths are not sorted?
- We call it *sift-up*



Binary Heap

`insert` 2

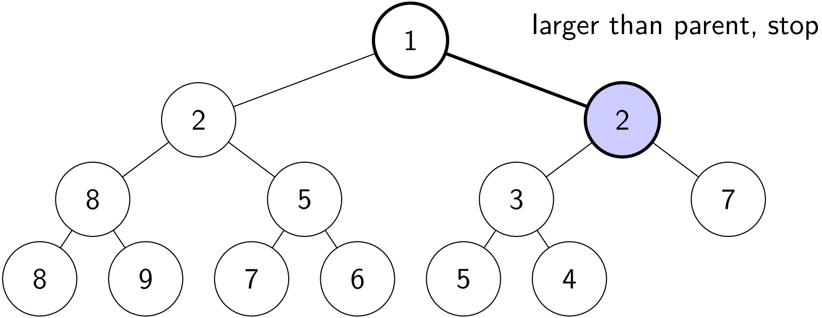
- Add new node at the bottom
- Heap property violated
- How many root-to-leaf paths are not sorted?
- We call it *sift-up*



Binary Heap

`insert` 2

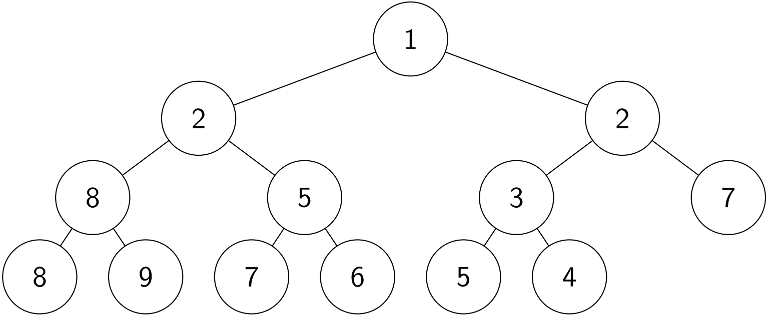
- Add new node at the bottom
- Heap property violated
- How many root-to-leaf paths are not sorted?
- We call it *sift-up*



Binary Heap

`insert` 2

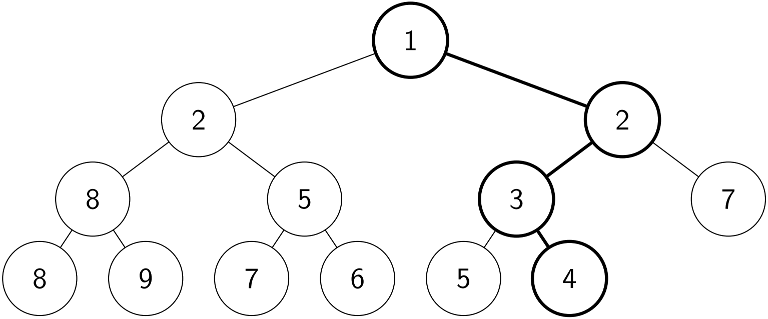
- Add new node at the bottom
- Heap property violated
- How many root-to-leaf paths are not sorted?
- We call it *sift-up*
- Done?



Binary Heap

Proof

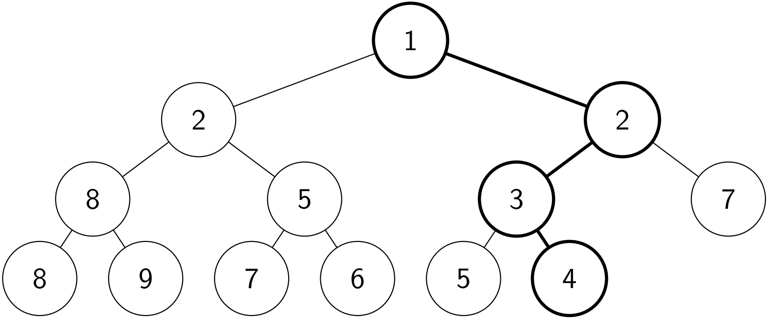
- The path on which we performed sift-up must be sorted
- Sift-up modified the *prefix*es of some other root-to-leaf paths
- Those *prefix*es must be sorted
- Those *prefix*es could only become smaller after sift-up
- So, all root-to-leaf paths are sorted



Binary Heap

Time complexity

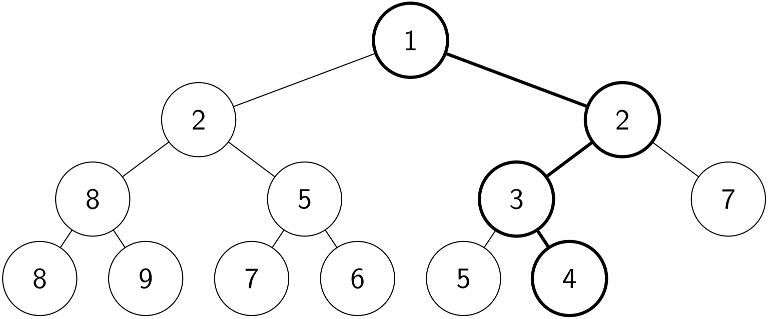
- Maximum number of swaps $= h = O(\log N)$



Binary Heap

`get min`

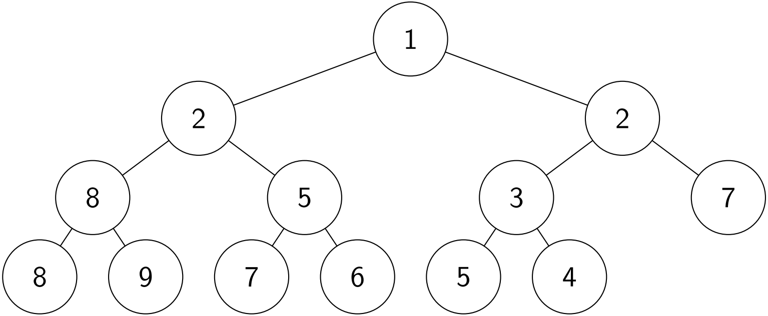
- Simply return root
- $O(1)$



Binary Heap

`remove min`

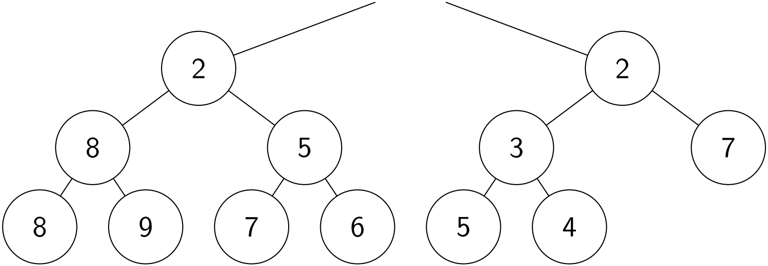
- Just remove it?



Binary Heap

`remove min`

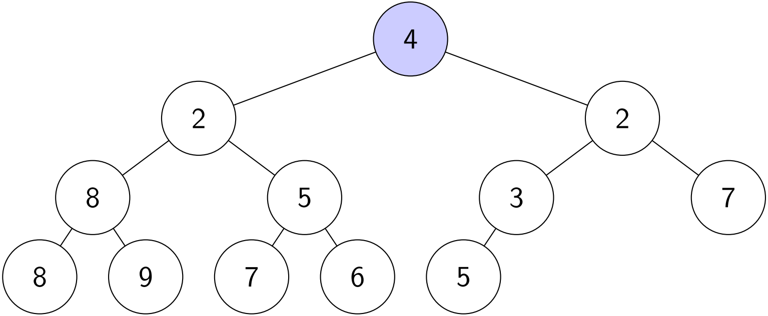
- Just remove it?
- Shape property violated
- Let's replace it with the bottom node



Binary Heap

`remove min`

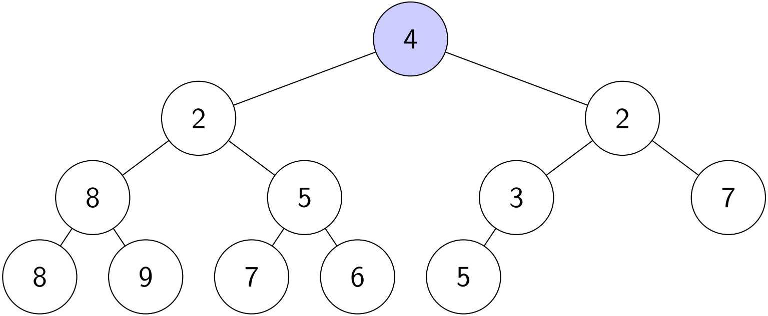
- Heap property violated
- How many root-to-leaf paths are not sorted?



Binary Heap

`remove min`

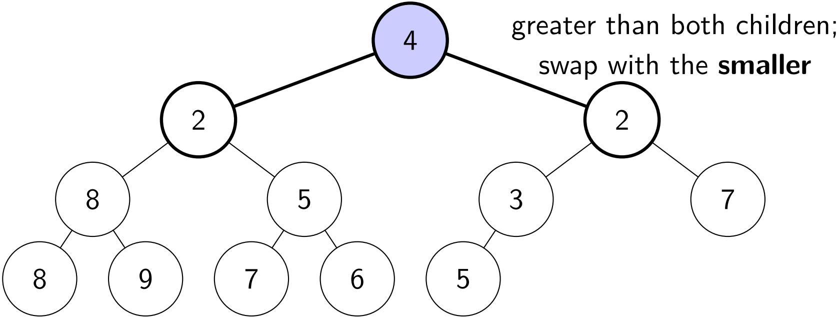
- Heap property violated
- How many root-to-leaf paths are not sorted?
- Perform *sift-down*



Binary Heap

`remove min`

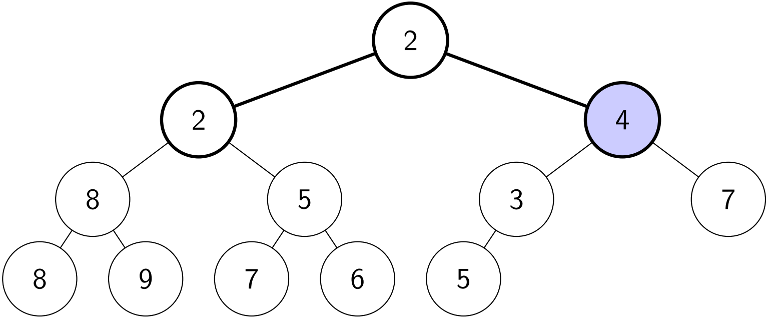
- Heap property violated
- How many root-to-leaf paths are not sorted?
- Perform *sift-down*



Binary Heap

`remove min`

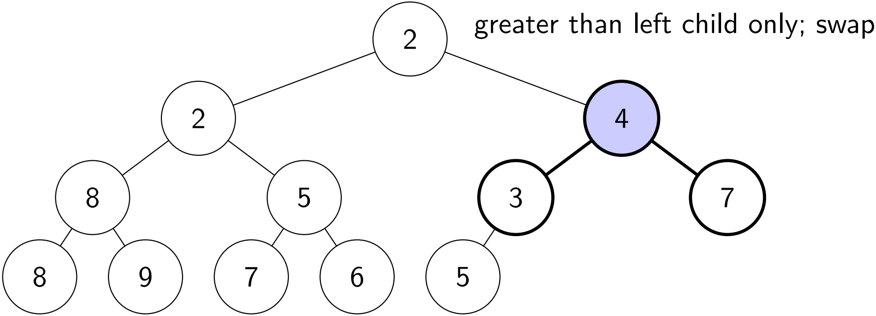
- Heap property violated
- How many root-to-leaf paths are not sorted?
- Perform *sift-down*



Binary Heap

`remove min`

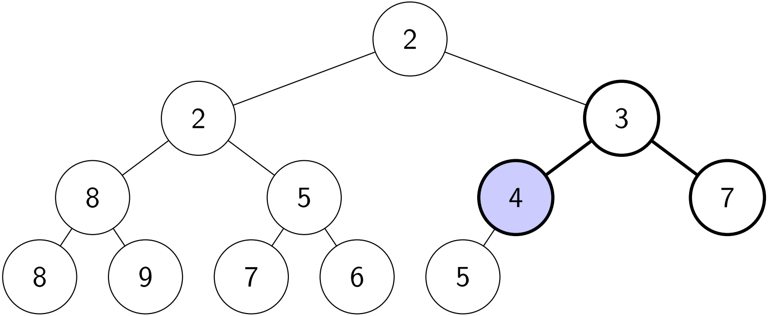
- Heap property violated
- How many root-to-leaf paths are not sorted?
- Perform *sift-down*



Binary Heap

`remove min`

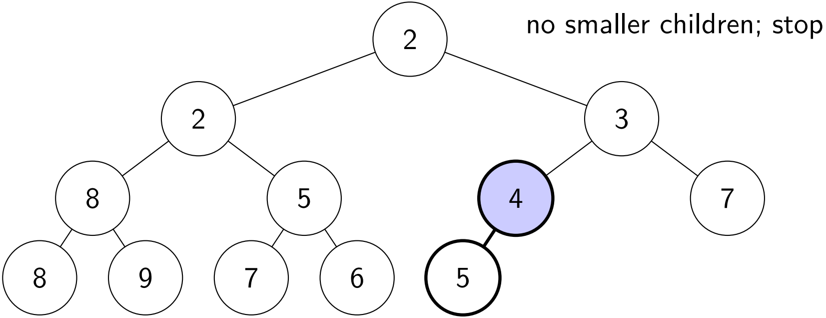
- Heap property violated
- How many root-to-leaf paths are not sorted?
- Perform *sift-down*



Binary Heap

`remove min`

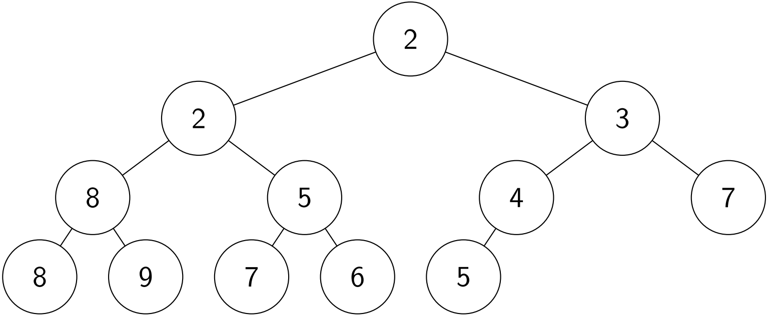
- Heap property violated
- How many root-to-leaf paths are not sorted?
- Perform *sift-down*



Binary Heap

`remove min`

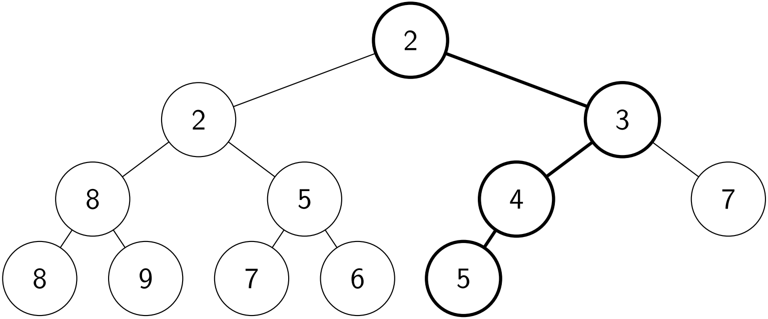
- Heap property violated
- How many root-to-leaf paths are not sorted?
- Perform *sift-down*
- Done?



Binary Heap

Proof

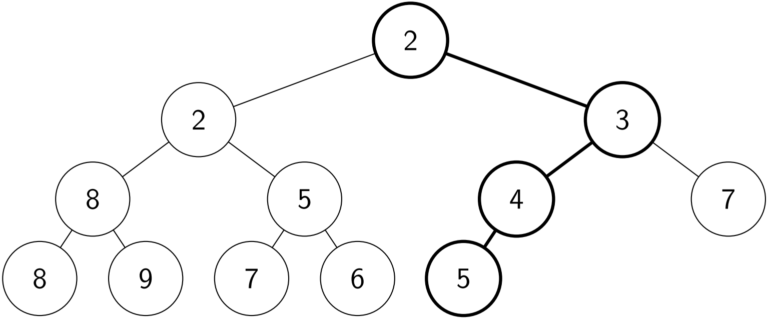
- The path on which we performed sift-down must be sorted
- Sift-down modified the *prefix*es of some root-to-leaf paths
- Those *prefix*es must be sorted
- Those *prefix*es could not become larger than the postfixes
- So, all root-to-leaf paths are sorted



Binary Heap

Time complexity

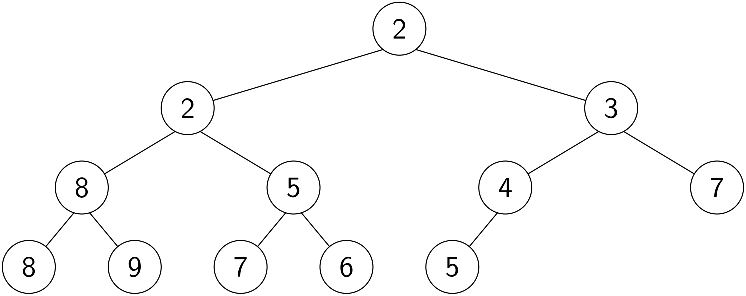
- Maximum number of swaps $= h = O(\log N)$



Binary Heap

Implementation

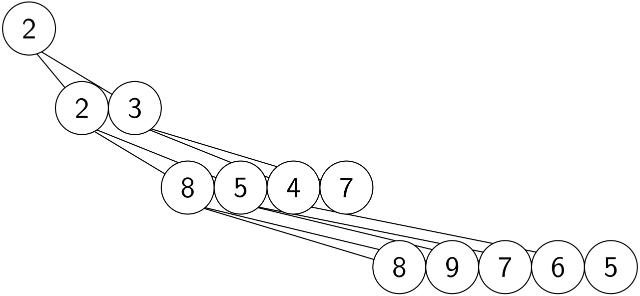
- plain array



Binary Heap

Implementation

- plain array

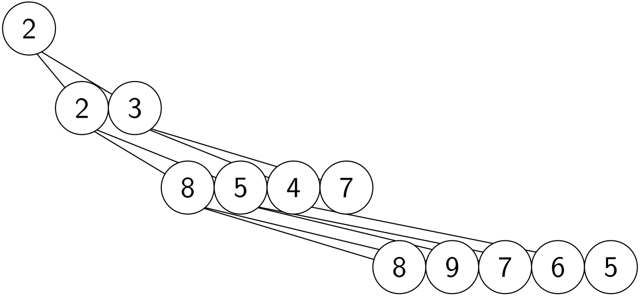


Binary Heap

Implementation

- plain array

- parent i
- left child $2i$
- right child $2i + 1$



	2	2	3	8	5	4	7	8	9	7	6	5
0	1	2	3	4	5	6	7	8	9	10	11	12

Heapsort

1. Build heap naively: $O(N \log N)$
2. Repeatedly remove min: $O(N \log N)$

Any similarity between heapsort and selection sort?

Heapsort

1. Build heap naively: $O(N \log N)$
2. Repeatedly remove min: $O(N \log N)$

Any similarity between heapsort and selection sort?

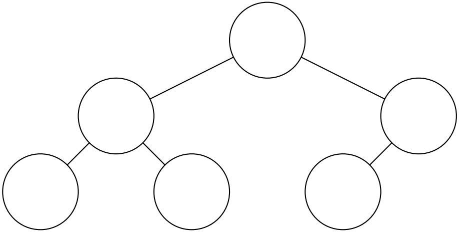
Heapsort vs Quicksort

- In reality, the computer reads not only one memory slot at a time, but a chunk of contiguous memory into cache
- Heapsort does not operate on contiguous memory, introducing cache miss
- Quicksort operate on contiguous memory, benefits from cache

Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs

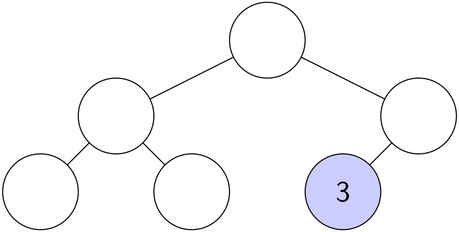
3	2	1	4	6	7



Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs

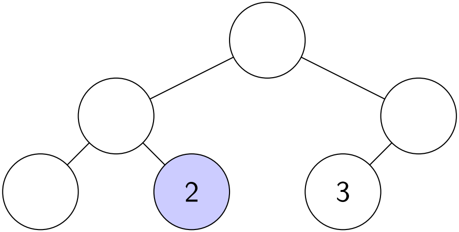
2	1	4	6	7



Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs

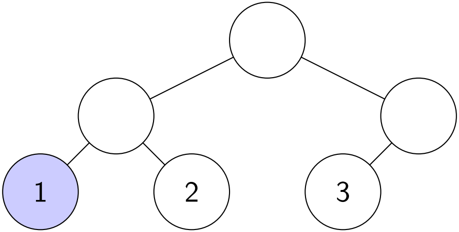
1	4	6	7



Build binary heap in linear time

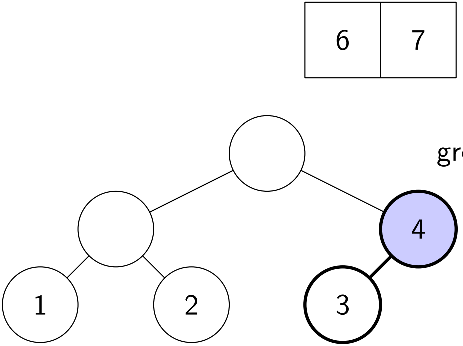
- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs

4	6	7



Build binary heap in linear time

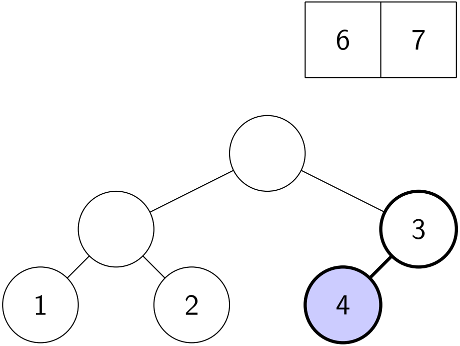
- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs



greater than left child; swap

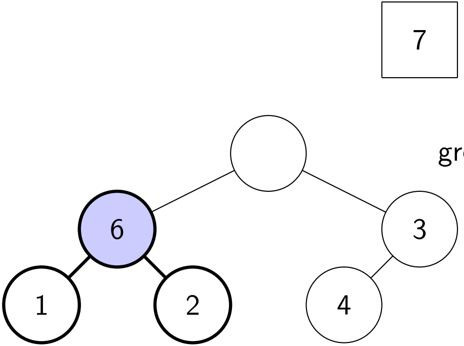
Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs



Build binary heap in linear time

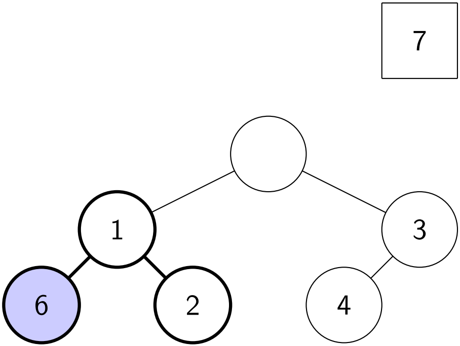
- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs



greater than left child; swap

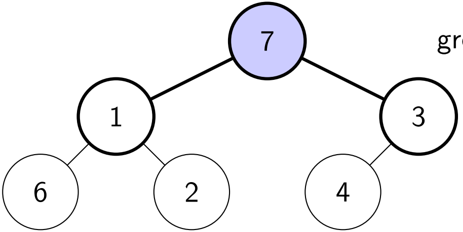
Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs



Build binary heap in linear time

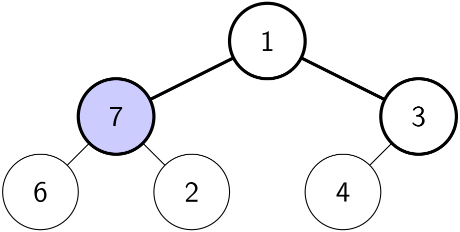
- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs



greater than left child; swap

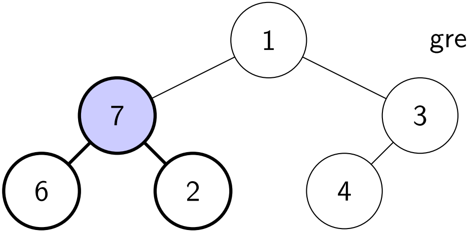
Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs



Build binary heap in linear time

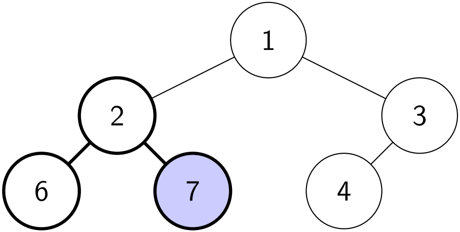
- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs



greater than right child; swap

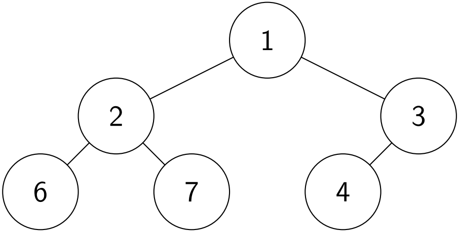
Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs



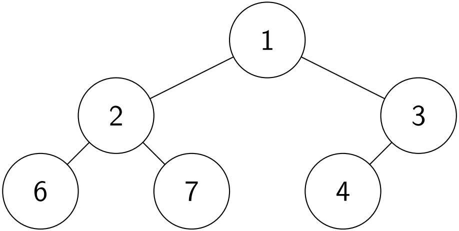
Build binary heap in linear time

- Create the shape first
- Add elements in the lowest levels first
- Perform sift-downs



Build binary heap in linear time

	layer	#swaps	#nodes
(top)	1^{st}	h	1
	2^{nd}	$h-1$	2
	3^{rd}	$h-2$	4
		$\dots$	
	h^{th}	1	2^{h-1}
(bottom)	$(h+1)^{th}$	0	2^h



Build binary heap in linear time

	layer	#swaps	#nodes
(top)	1^{st}	h	1
	2^{nd}	$h-1$	2
	3^{rd}	$h-2$	4
		$\dots$	
	h^{th}	1	2^{h-1}
(bottom)	$(h+1)^{th}$	0	2^h

$$\text{Total \#swaps} = \sum_{i=0}^{h} 2^{h-i} i = 2^h \sum_{i=0}^{h} \frac{i}{2^i} \le 2^h \sum_{i=0}^{\infty} \frac{i}{2^i} = 2^h(2) = O(N)$$

because

$$\sum_{i=0}^{\infty} \frac{i}{2^i} = \sum_{i=0}^{\infty} \frac{i+1}{2^{i+1}} = \sum_{i=0}^{\infty} \frac{i}{2^{i+1}} + \sum_{i=0}^{\infty} \frac{1}{2^{i+1}} = \frac{1}{2}\sum_{i=0}^{\infty} \frac{i}{2^i} + 1$$

$$\Rightarrow \sum_{i=0}^{\infty} \frac{i}{2^i} = 2$$

Build binary heap in linear time

- Time complexity: $O(N)$
- Does that mean we can sort an array in $O(N)$ time?

Build binary heap in linear time

- Time complexity: $O(N)$
- Does that mean we can sort an array in $O(N)$ time?
- No, removing all minimum still need $O(N \log N)$

Merging binary heaps

- Can we merge two binary heaps in $O(N \log N)$ time?

Merging binary heaps

- Can we merge two binary heaps in $O(N \log N)$ time?
  insert all elements from one to another
- Can we merge two binary heaps in $O(N)$ time?

Merging binary heaps

- Can we merge two binary heaps in $O(N \log N)$ time?
  insert all elements from one to another
- Can we merge two binary heaps in $O(N)$ time?
  concatenate two arrays and build heap
- Can we merge faster?
- Yes, but very complicated
- Better use alternatives such as *Leftist heap* or *Binomial heap*

C++ Standard Template Library

- The afore introduced binary heap was min-heap
- `priority_queue` is max-heap

```
#include <queue>
...
std::priority_queue<int> q;
q.push(4); q.push(7); q.push(3);
int x = q.top();       //x = 7
q.pop(); x = q.top(); //x = 4
```

Problem Description

N operations:

- `insert` X
- `exists` X
- `remove` X
- `get min`

Constraints:

- $1 \leq N \leq 10000$
- $1 \leq X \leq 10^{18}$

Input

```
6
insert 12
insert 39
insert 74
exists 74
remove min
get min
```

Output

```
yes
39
```

Binary Search Tree

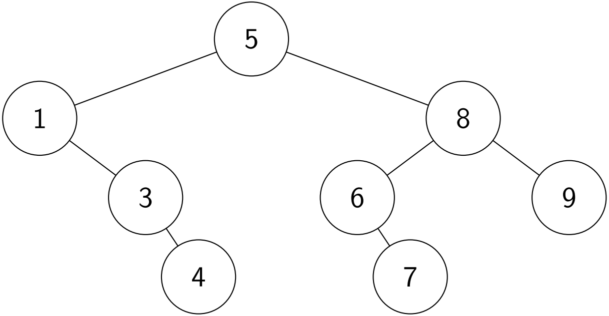
Binary tree

Every node $>$ its left descendants

Every node $<$ its right descendants

Do we get a sorted array in

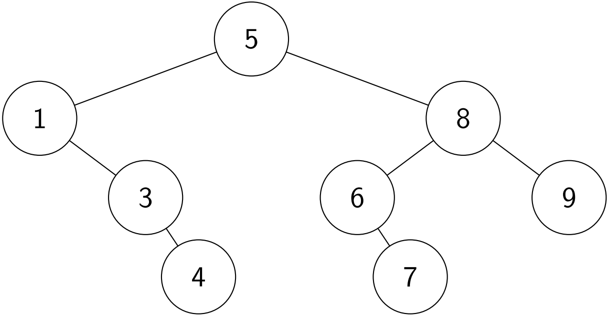
- pre-order traversal?
- in-order traversal?
- post-order traversal?



Binary Search Tree

`exists` 6

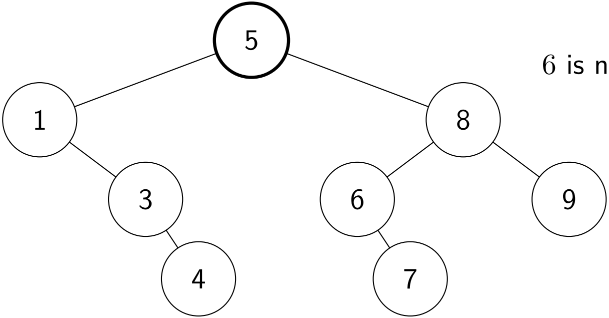
- Binary search from root



Binary Search Tree

`exists` 6

- Binary search from root

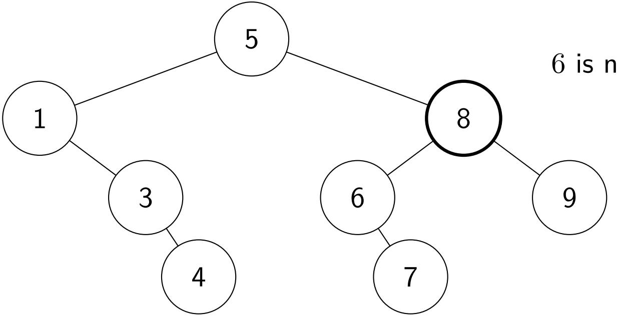


6 is not here; go right

Binary Search Tree

`exists` 6

- Binary search from root

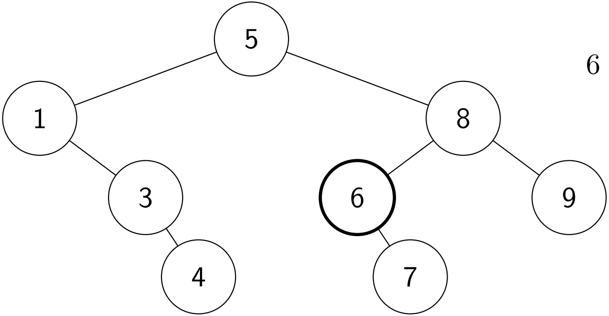


6 is not here; go left

Binary Search Tree

`exists` 6

- Binary search from root

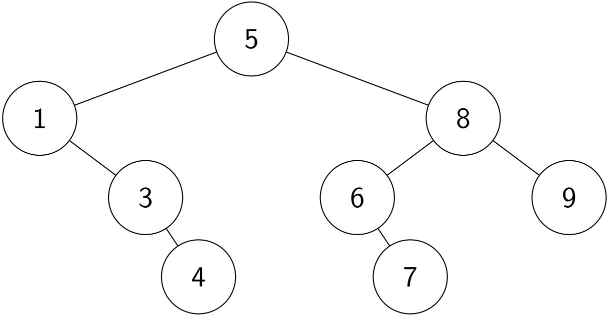


6 is here; done

Binary Search Tree

`exists` 6

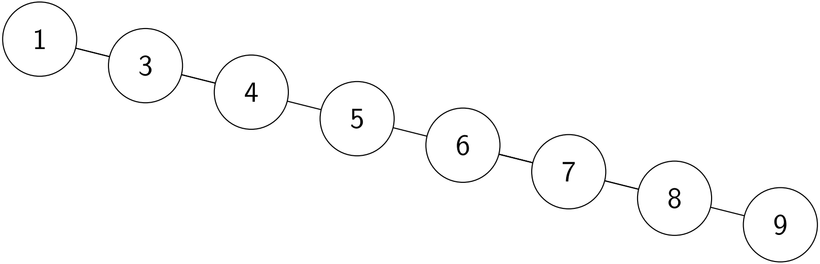
- Binary search from root
- Each iteration we discard half of the subtree
- Time complexity $= O(\log N)$ in average



Binary Search Tree

`exists` 6

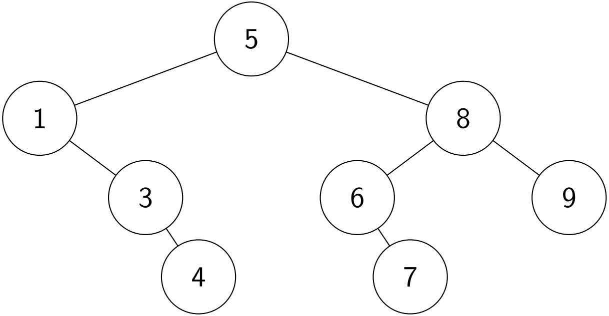
- Binary search from root
- Each iteration we discard half of the subtree
- Time complexity $= O(\log N)$ in average
- Time complexity $= O(N)$ in worst case



Binary Search Tree

`insert` 2

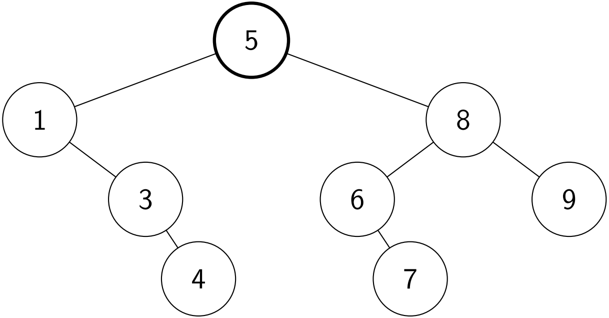
- Binary search a place to insert



Binary Search Tree

`insert` 2

- Binary search a place to insert

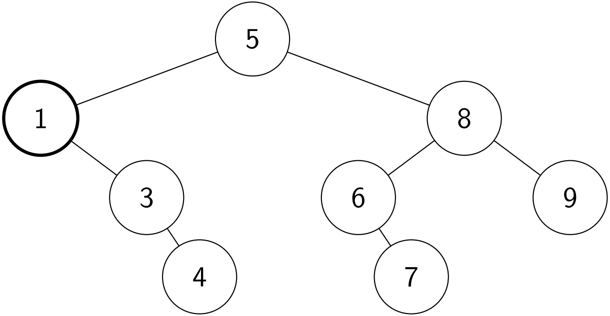


go left

Binary Search Tree

`insert` 2

- Binary search a place to insert

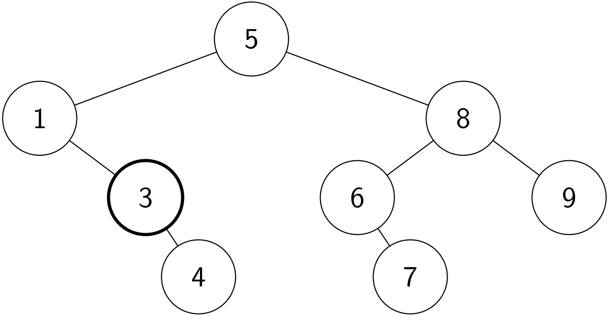


go right

Binary Search Tree

`insert` 2

- Binary search a place to insert

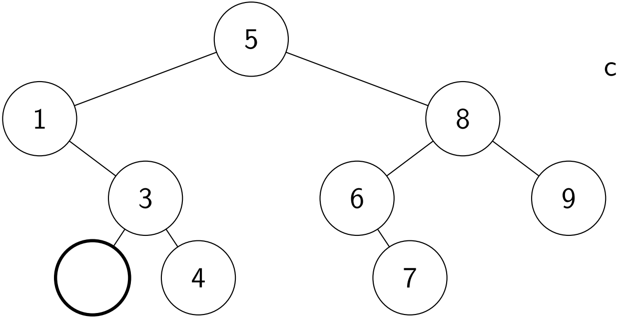


go left

Binary Search Tree

`insert` 2

- Binary search a place to insert

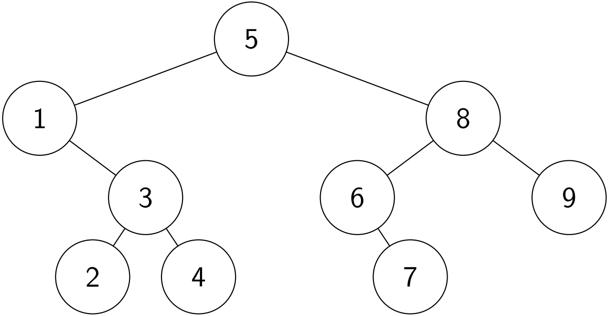


create node

Binary Search Tree

`insert` 2

- Binary search a place to insert
- Each iteration we discard half of the subtree
- Time complexity $= O(\log N)$ in average

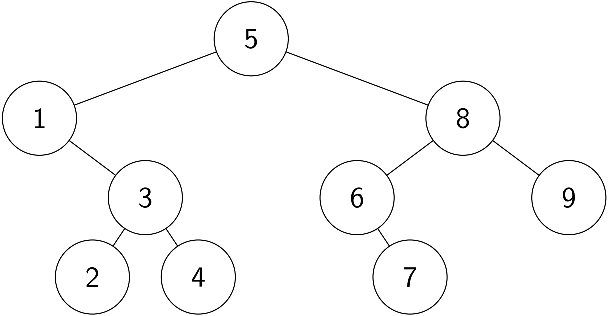


insert

Binary Search Tree

`insert` 2

- Binary search a place to insert
- Each iteration we discard half of the subtree
- Time complexity $= O(\log N)$ in average
- Time complexity $= O(N)$ in worst case

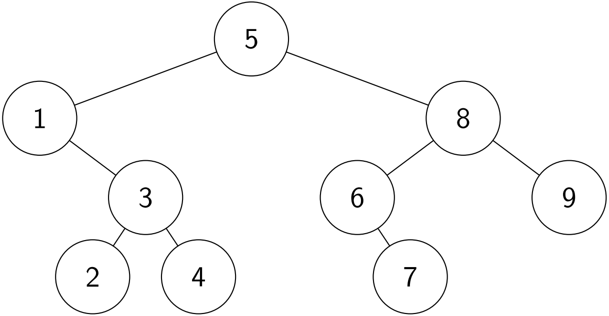


Binary Search Tree

`remove` 8

`remove` 5

- Binary search the node
- Replace it with the greatest node less than it

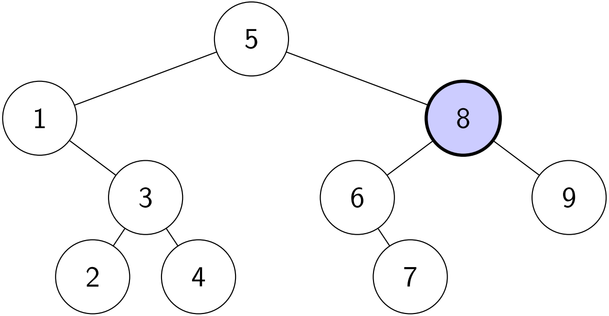


Binary Search Tree

`remove` 8

`remove` 5

- Binary search the node
- Replace it with the greatest node less than it



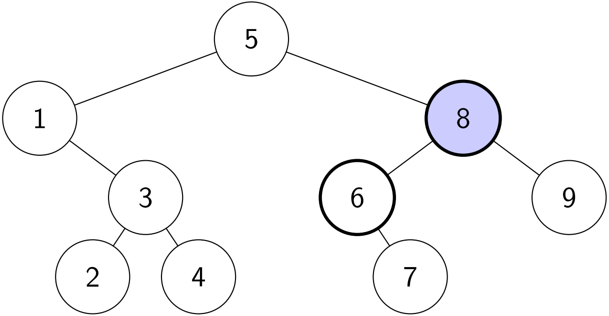
go left

Binary Search Tree

`remove` 8

`remove` 5

- Binary search the node
- Replace it with the greatest node less than it



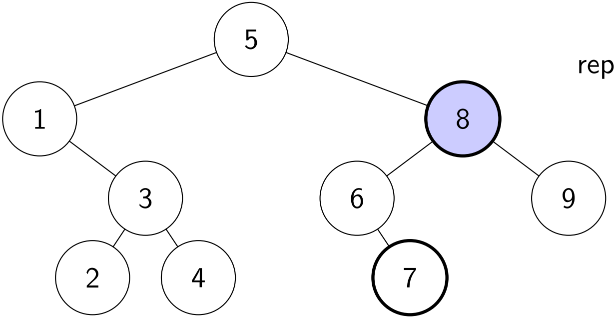
go right

Binary Search Tree

`remove` 8

`remove` 5

- Binary search the node
- Replace it with the greatest node less than it



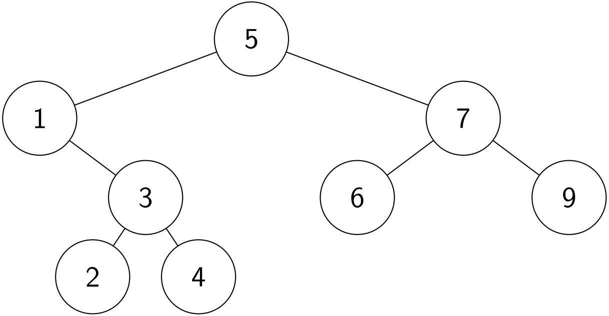
replace 8 with 7

Binary Search Tree

`remove` 8

`remove` 5

- Binary search the node
- Replace it with the greatest node less than it

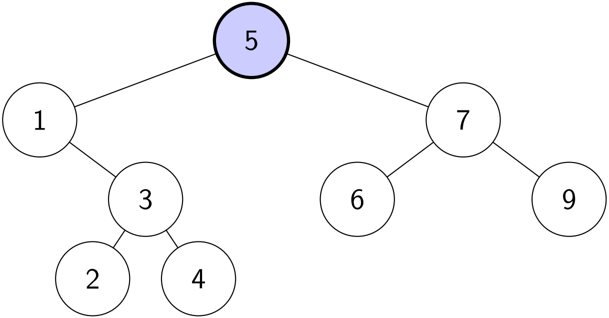


Binary Search Tree

`remove` 8

`remove` 5

- Binary search the node
- Replace it with the greatest node less than it



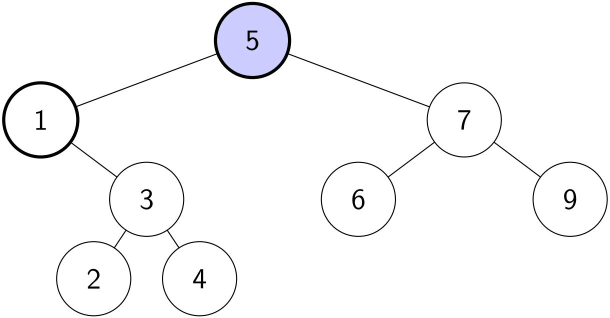
go left

Binary Search Tree

`remove` 8

`remove` 5

- Binary search the node
- Replace it with the greatest node less than it



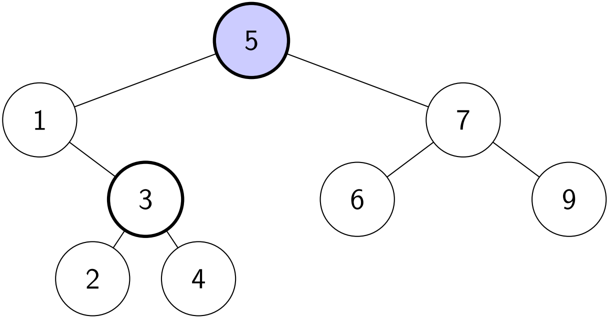
go right

Binary Search Tree

`remove` 8

`remove` 5

- Binary search the node
- Replace it with the greatest node less than it



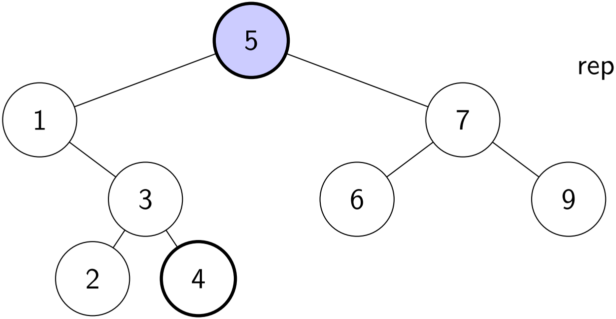
go right

Binary Search Tree

`remove` 8

`remove` 5

- Binary search the node
- Replace it with the greatest node less than it



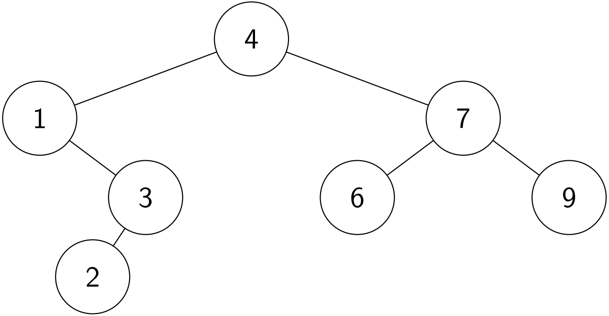
replace 5 with 4

Binary Search Tree

`remove` 8

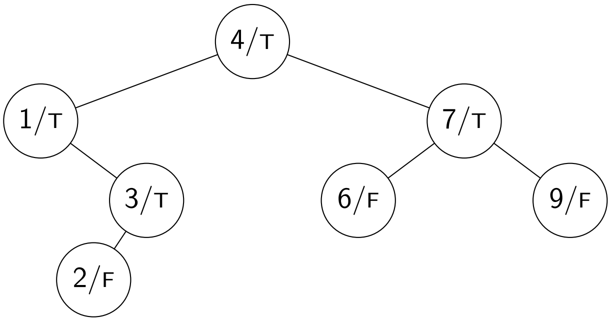
`remove` 5

- Binary search the node
- Replace it with the greatest node less than it
- Locate the node $= O(\log N)$ in average
- Locate the smaller node $= O(\log N)$ in average
- Time complexity $= O(\log N)$ in average
- Time complexity $= O(N)$ in worst case



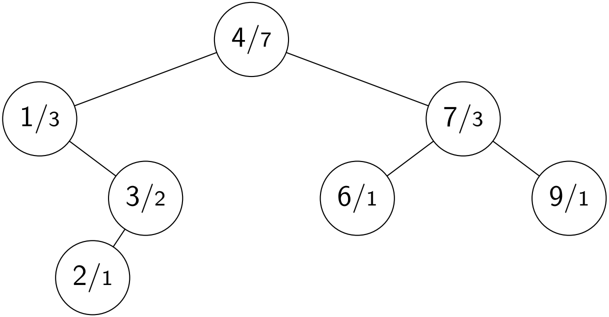
Augmented Binary Search Tree

- We can attach *value*s to nodes, e.g.
  - existence
- Nodes are sorted by their *key*s
- How does it affect `insert` and `remove`?
- Do we still need `remove`?



Augmented Binary Search Tree

- We can attach *values* to nodes, e.g.
  - existence
  - subtree size
- Nodes are sorted by their *keys*
- How does it affect `insert` and `remove`?
- Can we find the k^{th} largest element?

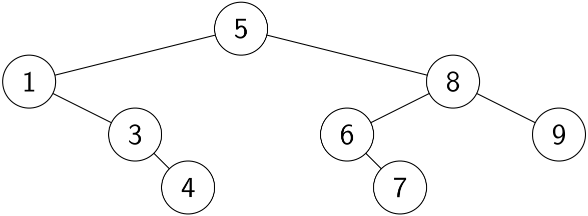


Binary Search Tree

- It is easy to generate a worst case testcase
- How to get rid of the $O(N)$ worst case?
- Use variants of BST
  - AVL tree
  - Red-black tree
  - AA tree
  - Treap
  - Size-balanced tree
  - Splay tree

Implementation

```
int root, value[10000], left[10000], right[10000];
```



	0	1	2	3	4	5	6	7		
value	1	6	5	8	3	9	7	4		
left			0	1						
right	4	6	3	5	7					

Implementation

or, if you know C++ and pointer very well,

```
struct BST {
  int value;
  BST *left, *right;
} *root, memory[10000], memory_i;
```

Avoid `malloc()` because it is slow

C++ Standard Template Library

- `set`, `map`, `multiset`, `multimap`
- Red-black tree internally
- worst case $O(\log N)$

```
#include <set>
...
std::set<int> s;
s.insert(456);
int x = s.count(456); //1
x.erase(456);
```

```
#include <map>
...
std::map<int, int> m;
m[456] = 123;
int x = m[456]; //123
m.erase(456);
```

Practice problems

- 01090 Diligent
- 01019 Addition II
- 30107 What is the Median?
- M0811 Alice's Bookshelf
- M0913 Is
- AP121 Dispatching

启发式合并

- Two literal meanings:
  - Insightful merging
  - Open-up merging (not this one)
- Similar to *Union by Rank* in disjoint-set union (Data Structures (III))
- Seems there is not a commonly used English term
- Problem: need to merge N items from small binary heaps into a large binary heap
- Idea: when merging two heaps, insert the smaller one into the larger one
- Time complexity: $O(N \log^2 N)$

启发式合并

- Consider an item X
- Initially X belongs to a heap of size 1
- Whenever the heap containing X is merged into a larger heap, the size of the heap is at least doubled
- So X can be inserted into another heap for at most $O(\log N)$ times
- Each of the N items can be inserted into another heap for at most $O(\log N)$ times
- Total number of heap insertions $= O(N \log N)$
- Time complexity $= O(N \log^2 N)$