Range Minimum Queries and Lowest Common Ancestor

Inge Li Gørtz

Range Minimum Queries and Lowest Common Ancestor

- Range Minimum Queries (RMQ) and Lowest Common Ancestor (LCA)
- RMQ
  - Simple solutions
  - Better solution
  - 2-level solution
- Reduction between RMQ and LCA
- Dynamic RMQ

Range Minimum Queries

- Range minimum query problem. Preprocess array A[1…n] of integers to support
  - RMQ(i,j): return the (entry of) minimum element in A[i…j].

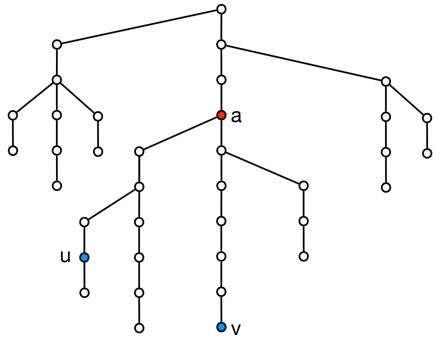
0	1	2	3	4	5	6	7	8	9
1	7	12	8	2	5	1	4	8	3

  - RMQ(2,5) = 2 (index 4)

- Basic (extreme) solutions
  - Linear search:
    - Space: O(n). Only keep array (no extra space)
    - Time: O(j-i) = O(n)
  - Save all possible answers: Precompute and save all answers in a table.
    - Space: $O(n^2)$ pairs => $O(n^2)$ space
    - Time: O(1)

Lowest Common Ancestor

- Lowest common ancestor problem. Preprocess rooted tree T with n nodes to support
  - LCA(u,v): return the lowest common ancestor of u and v.



LCA(u,v) = a

Lowest Common Ancestor

- Basic (extreme) solutions
  - Linear search: Follow paths to root and mark when you visit a node.
    - Space: O(n). Only keep tree (no extra space)
    - Time: O(depth of tree) = O(n)
  - Save all possible answers: Precompute and save all answers in a table.
    - Space: $O(n^2)$ pairs => $O(n^2)$ space
    - Time: O(1)

RMQ and LCA

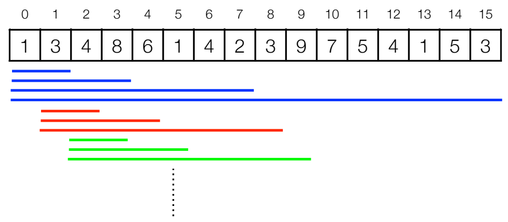
- Outline.
  - Can solve both RMQ and LCA in linear space and constant time.
    - First solution to RMQ
    - Solution to a special case of RMQ.
    - See that RMQ and LCA are equivalent (can reduce one to the other both ways).

RMQ

Sparse table solution

RMQ: Sparse table solution

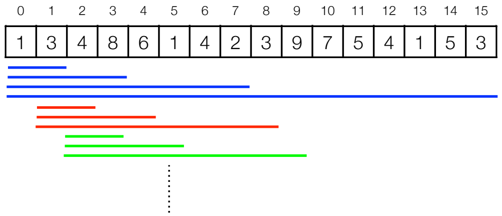
- Save the result for all intervals of length a power of 2.



	0	1	2	3	4
0	1	1	1	1	1
1	3	3	3	1	
2	4	4	1	1	
3	8	6	1	1	
4	6	1	1	1	
5	1	1	1	1	
6	4	2	2	1	
7	2	2	2	1	
8	3	3	3	1	
9	9	7	4		
10	7	5	1		
11	5	4	1		
12	4	1	1		
13	1	1			
14	5	3			
15	3				

RMQ: Sparse table solution

- Save the result for all intervals of length a power of 2.

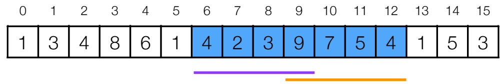


	0	1	2	3	4
0	1	1	1	1	1
1	3	3	3	1	
2	4	4	1	1	
3	8	6	1	1	
4	6	1	1	1	
5	1	1	1	1	
6	4	2	2	1	
7	2	2	2	1	
8	3	3	3	1	
9	9	7	4		
10	7	5	1		
11	5	4	1		
12	4	1	1		
13	1	1			
14	5	3			
15	3				

- Space: O(n log n)

RMQ: Sparse table solution

- Save the result for all intervals of length a power of 2.



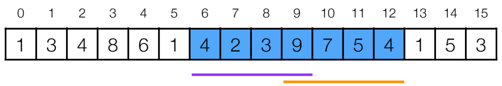
RMQ(6, 12) = ?

	0	1	2	3	4
0	1	1	1	1	1
1	3	3	3	1	
2	4	4	1	1	
3	8	6	1	1	
4	6	1	1	1	
5	1	1	1	1	
6	4	2	2	1	
7	2	2	2	1	
8	3	3	3	1	
9	9	7	4		
10	7	5	1		
11	5	4	1		
12	4	1	1		
13	1	1			
14	5	3			
15	3				

- Space: O(n log n)

RMQ: Sparse table solution

- Save the result for all intervals of length a power of 2.



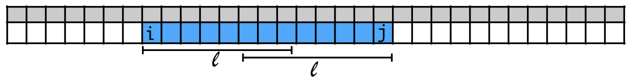
RMQ(6, 12) = min(RMQ(6,9), RMQ(9,12)) = min(2,4) = 2

	0	1	2	3	4
0	1	1	1	1	1
1	3	3	3	1	
2	4	4	1	1	
3	8	6	1	1	
4	6	1	1	1	
5	1	1	1	1	
6	4	2	2	1	
7	2	2	2	1	
8	3	3	3	1	
9	9	7	4		
10	7	5	1		
11	5	4	1		
12	4	1	1		
13	1	1			
14	5	3			
15	3				

- Space: O(n log n)

RMQ: Sparse table solution

- Query:



  - Any interval is the union of two power of 2 intervals.
    - k largest number such that $2^k \leq j - i + 1$.
  - Lookup results for the two intervals and take minimum.
- Time: O(1)
- Space: O(n log n)
- Preprocesing time: O(n log n)
  - To compute results for length 2^i use results for length 2^{i-1}.

±1RMQ

RMQ: Linear space

- Consider ±1RMQ: consecutive entries differ by 1.

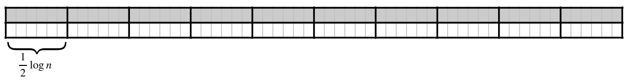
0	1	2	3	4	5	6	7	8	9	10	11	12
4	5	6	5	4	3	2	3	2	3	4	5	4

- 2-level solution: Combine
  - O(n log n) space, O(1) time
  - $O(n^2)$ space, O(1) time.

    $\Downarrow$
  - O(n) space, O(1) time.

±1RMQ

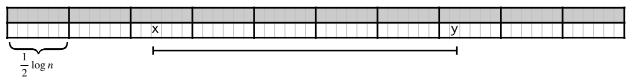
- Divide A into blocks of size $\frac{1}{2}\log n$



- 2-level data structure:
  - Sparse table on blocks
  - Tabulation inside blocks.

±1RMQ

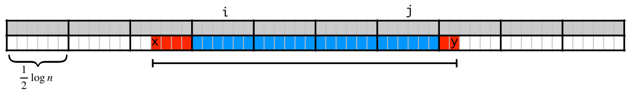
- Divide A into blocks of size $\frac{1}{2}\log n$



- 2-level data structure:
  - Sparse table on blocks
  - Tabulation inside blocks.

±1RMQ

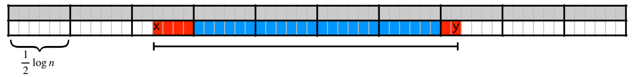
- Divide A into blocks of size $\frac{1}{2}\log n$



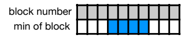
- 2-level data structure:
  - Sparse table on blocks
  - Tabulation inside blocks.

- RMQ(x,y) = min{ RMQ on blocks i to j,
  RMQ inside block i-1,
  RMQ inside block j+1 }.

±1RMQ: Data structure on blocks

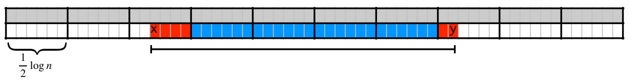


- Two new arrays.
  - Array A': minimum from each block
  - B: position in A where $A'[i]$ occurs.

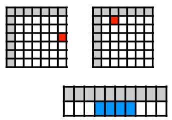


- Sparse table data structure on A'.
- Space: $O(|A'|\log|A'|) = O\left(\frac{n}{\log n}\cdot\log\frac{n}{\log n}\right) = O(n)$.
- Time: $O(1)$

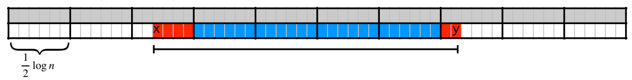
±1RMQ: Data structure inside blocks



- Precompute and save all answers for each block.
- Gives solution using
  - Space:

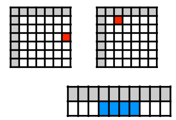


±1RMQ: Data structure inside blocks



- Precompute and save all answers for each block.
- Gives solution using
  - Space: O(n) + space for precomputed tables.
  - Time: O(1) + O(1) + O(1) = O(1).

  2 table lookups → O(1); sparse table → O(1); min{•,•,•} → O(1)

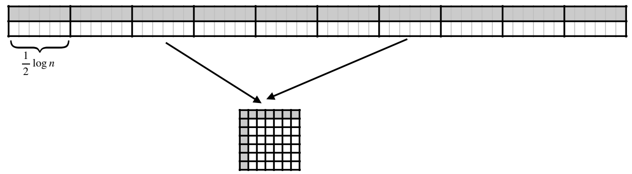


±1RMQ: Storing the tables

- Naively: $\log^2 n$ for each table => n log n space. 🙁
- Observation: If X[i] = Y[i] + c then all RMQ answers are the same for X and Y.
  - X = [7, 6, 5, 6, 5, 4]
  - Y = [3, 2, 1, 2, 1, 0]
- Normalize blocks:
  - X = [0, -1, -2, -1, -2, -3] = Y
- Normalized block described by sequence of +1s and -1s:
  - X = Y = -1, -1, +1, -1, -1.
- How many different normalized blocks are there?
  - length of sequence = $\frac{1}{2}\log n - 1$
  - #sequences = $2^{\frac{1}{2}\log n - 1} \leq \sqrt{n}$.

±1RMQ: Data structure inside blocks

- Precompute and save all answers for each normalized block.
- Size of a table: $O(\log^2 n)$
- For each block save which precomputed table it uses.



- Space: $O(\sqrt{n} \cdot \log^2 n) + O(n/\log n) = O(n)$
- Plugging into 2-level solution:
  - Space: O(n) + space for precomputed tables = O(n).

LCA and RMQ

RMQ and LCA

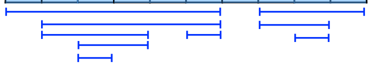
- We will show
  - RMQ → (reduces to) LCA → (reduces to) ±1RMQ

If there is a solution to LCA using s(n) space and t(n) time, then there is a solution to RMQ using O(s(n)) space and O(t(n)) time.

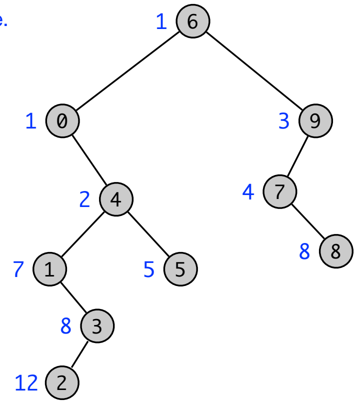
If there is a solution to ±1RMQ using s(n) space and t(n) time, then there is a solution to LCA using O(s(n)) space and O(t(n)) time.

RMQ to LCA

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



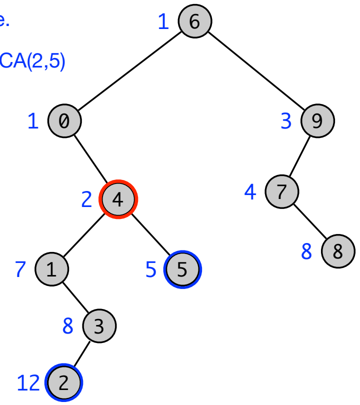
- Cartesian tree.



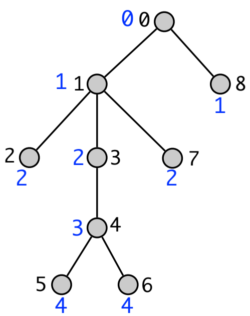
RMQ to LCA

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

- Cartesian tree.
- RMQ(2,5) = LCA(2,5)



LCA to ±1RMQ



- E =

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	1	2	1	3	4	5	4	6	4	3	1	7	1	0	8	0

- A =

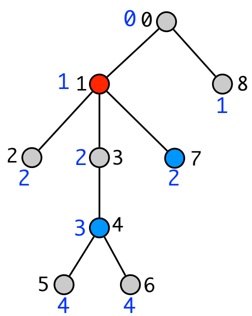
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	1	2	1	2	3	4	3	4	3	2	1	2	1	0	1	0

- R =

0	1	2	3	4	5	6	7	8
0	1	2	4	5	6	8	12	15

- E: Euler tour representation: preorder walk, write preorder number of node when met.
- A: depth of node node in E[i].
- R: first occurrence in E of node with preorder number i
- LCA(i, j) = E[RMQ_A(R[i], R[j])].

LCA to ±1RMQ



- LCA(4,7) = RMQ_A(5, 12).

- E =

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	1	2	1	3	4	5	4	6	4	3	1	7	1	0	8	0

- A =

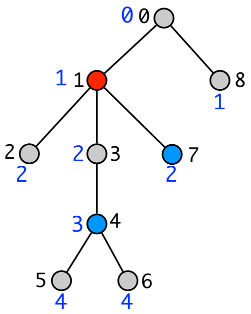
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	1	2	1	2	3	4	3	4	3	2	1	2	1	0	1	0

- R =

0	1	2	3	4	5	6	7	8
0	1	2	4	5	6	8	12	15

- E: Euler tour representation: preorder walk, write preorder number of node when met.
- A: depth of node node in E[i].
- R: first occurrence in E of node with preorder number i
- LCA(i, j) = E[RMQ_A(R[i], R[j])].

LCA to ±1RMQ



- LCA(4,7) = $RMQ_A(5, 12)$.

- E =

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	1	2	1	3	4	5	4	6	4	3	1	7	1	0	8	0

|E| = 2n

- A =

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
0	1	2	1	2	3	4	3	4	3	2	1	2	1	0	1	0

|A| = 2n

- R =

0	1	2	3	4	5	6	7	8
0	1	2	4	5	6	8	12	15

|R| = n

Space O(n):
- 3 tables
- ±1RMQ data structure on table of length 2n

- E: Euler tour representation: preorder walk, write preorder number of node when met.
- A: depth of node node in E[i].
- R: first occurrence in E of node with preorder number i
- LCA(i, j) = E[RMQ_A(R[i], R[j])].

RMQ and LCA

- Theorem. RMQ and LCA can be solved in O(n) space and O(1) query time.

Segment trees

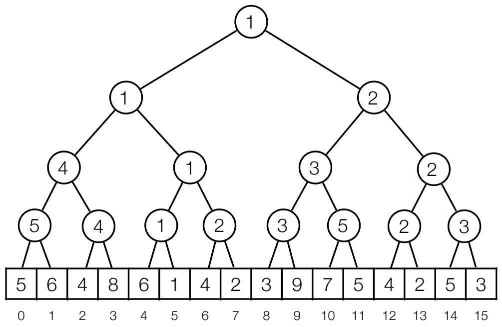
Dynamic Range Minimum Queries

Segment trees

- Dynamic RMQ: Support following operations.
  - Add(i, k): Set A[i] = A[i] + k   (k can be negative).
  - RMQ(i,j)

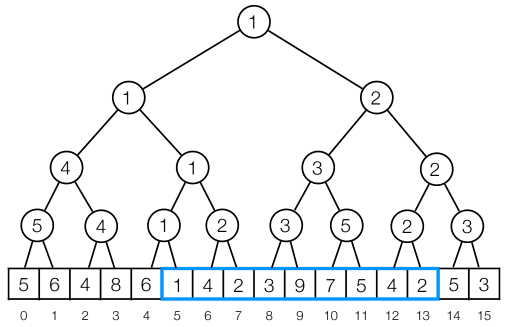
Segment trees

- Dynamic RMQ: Support following operations.
  - Add(i, k): Set A[i] = A[i] + k   (k can be negative).
  - RMQ(i,j)



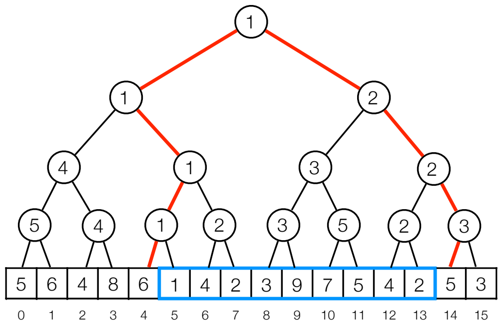
Segment trees

- Dynamic RMQ
  - RMQ(5,13) = ?



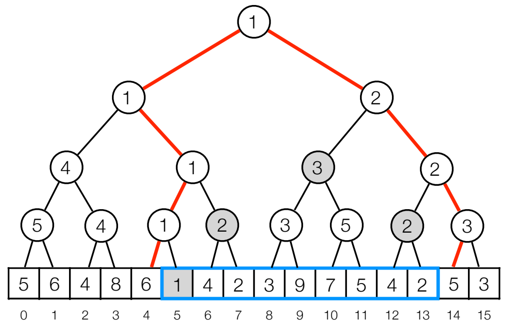
Segment trees

- Dynamic RMQ
  - RMQ(5,13) = ?



Segment trees

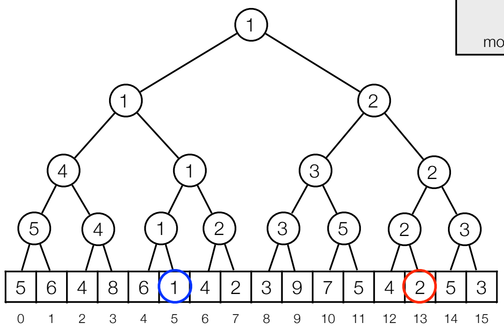
- Dynamic RMQ
  - RMQ(5,13) = ?



Segment trees

- Dynamic RMQ
  - RMQ(5,13) = INF

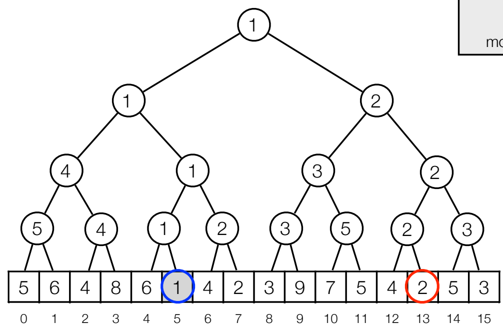
```
s = INF
while (a not right of b):
    if (a right child):
        s = min(s, tree[a])
        move a to the right
    if (b left child):
        s = min(s, tree[b])
        move b to the left
    move a and b to parents
```



Segment trees

- Dynamic RMQ
  - RMQ(5,13) = 1

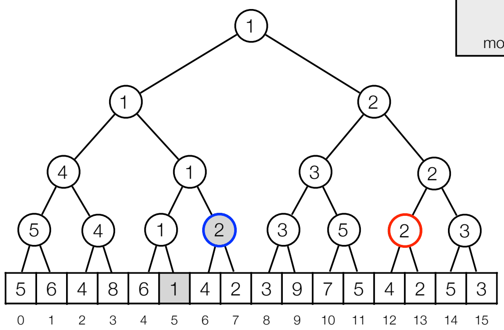
```
s = INF
while (a not right of b):
    if (a right child):
        s = min(s, tree[a])
        move a to the right
    if (b left child):
        s = min(s, tree[b])
        move b to the left
    move a and b to parents
```



Segment trees

- Dynamic RMQ
  - RMQ(5,13) = 1

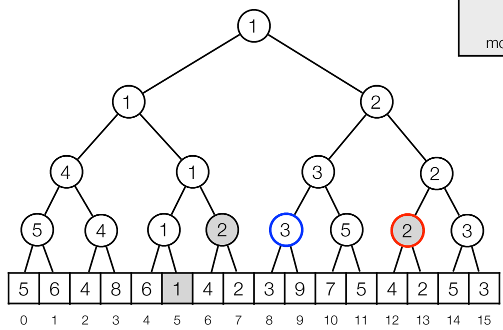
```
s = INF
while (a not right of b):
    if (a right child):
        s = min(s, tree[a])
        move a to the right
    if (b left child):
        s = min(s, tree[b])
        move b to the left
    move a and b to parents
```



Segment trees

- Dynamic RMQ
  - RMQ(5,13) = 1

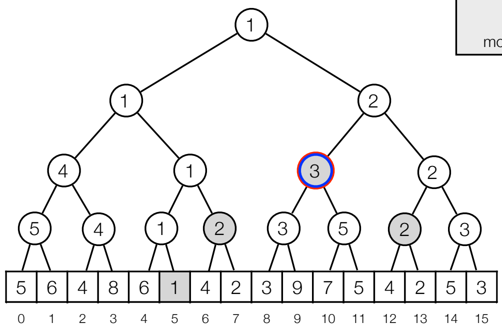
```
s = INF
while (a not right of b):
    if (a right child):
        s = min(s, tree[a])
        move a to the right
    if (b left child):
        s = min(s, tree[b])
        move b to the left
    move a and b to parents
```



Segment trees

- Dynamic RMQ
  - RMQ(5,13) = 1

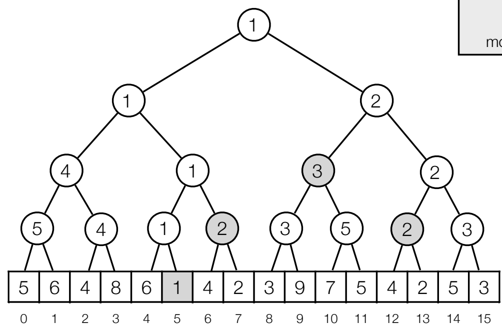
```
s = INF
while (a not right of b):
    if (a right child):
        s = min(s, tree[a])
        move a to the right
    if (b left child):
        s = min(s, tree[b])
        move b to the left
    move a and b to parents
```



Segment trees

- Dynamic RMQ
  - RMQ(5,13) = 1

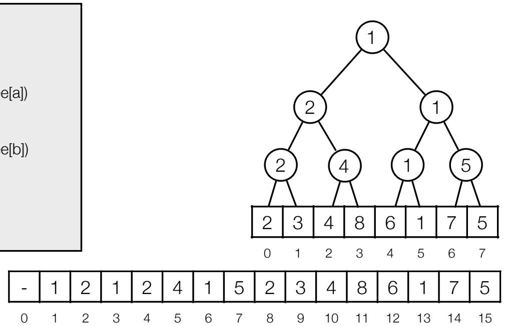
```
s = INF
while (a not right of b):
    if (a right child):
        s = min(s, tree[a])
        move a to the right
    if (b left child):
        s = min(s, tree[b])
        move b to the left
    move a and b to parents
```



Implementation

- Implement tree using heap layout in array of length 2n:
  - Root at position 1.
  - Children of node i at position 2i and 2i+1.

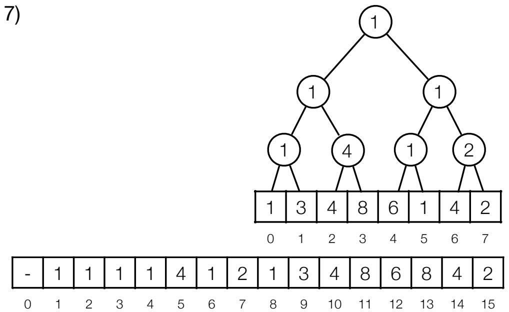
```
m = INFINITY
a += n, b+= n
while (a ≤ b):
    if (a % 2 == 1):
        m = min(m, tree[a])
        a += 1
    if (b % 2 == 0):
        m = min(m, tree[b])
        b -= 1
    a = ⌊a / 2⌋
    b = ⌊b / 2⌋
return m
```



Space: O(n)
Time: O(log n)

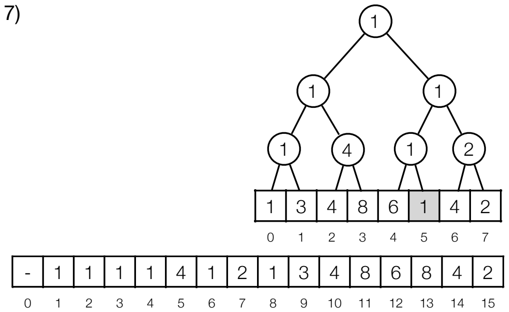
Updates

- Add(5, 7)



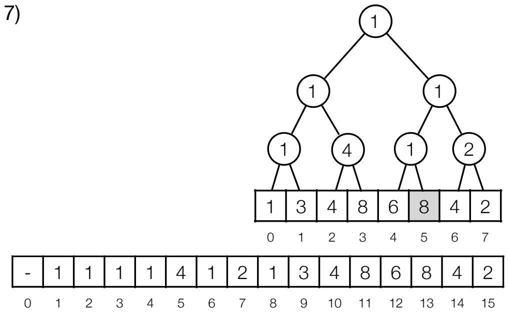
Updates

- Add(5, 7)



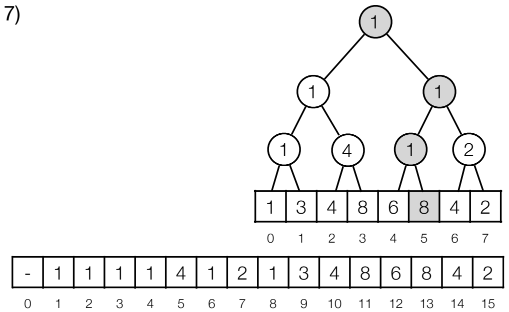
Updates

- Add(5, 7)



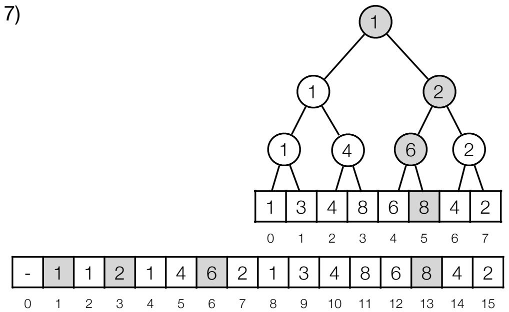
Updates

- Add(5, 7)



Updates

- Add(5, 7)




```
Add(i, k):
    i += n
    tree[i] += k
    i = ⌊i /2⌋
    while (i ≥ 1):
        tree[x] = min(tree[2*i], tree[2*i +1])
        i = ⌊i /2⌋
```

Time: O(log n)