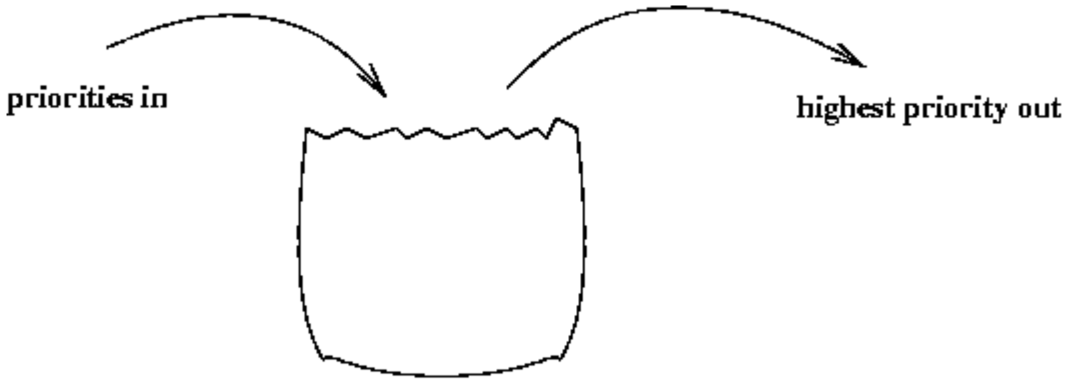
Data structures III

Agenda

1. ~~Binary search tree~~
2. Priority queue
3. Heap
4. Segment trees
5. ~~Hashtable~~

Priority queue



- A priority queue is a data structure for maintaining a set S of elements, each with an **associated value called a key.**

x0=Element:4

x1=Element:3

Priority queue operations

A priority queue supports the following operations:

- INSERT(S,x) inserts the element x into the set S, which we can write: S <- S union {x}
- MAXIMUM(S) returns the element in S with the largest key.
- EXTRACT-MAX(S) returns the element in S with the largest key and removes it.
- INCREASE-KEY(S,x,k) increases the value of x's key to k (assumed to be >= x's current key).

- INSERT({(element1,3)},(element2,4)) =

  {(element1,3)} **U** {(element2,4)} =

  S ={(element1,3),(element2,4)}
- MAXIMUM(S) = (element2,4)

  S ={(element1,3),(element2,4)}
- EXTRACT-MAX(S) = (element2,4)

  S ={(element1,3)}
- INCREASE-KEY(S,element1,4) =

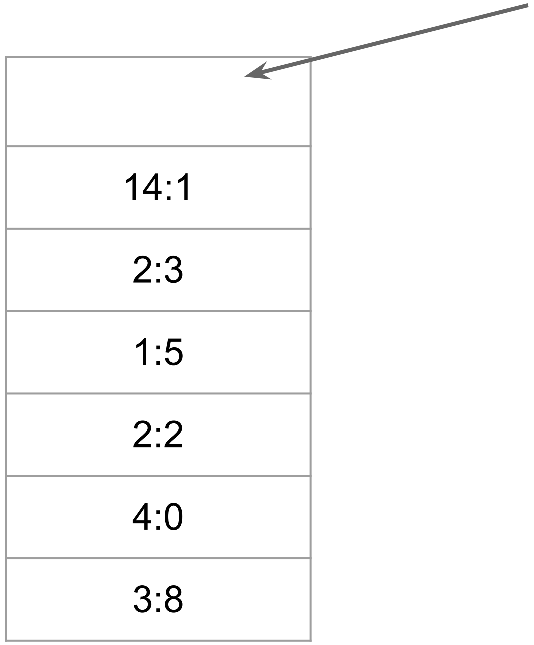
  S ={(element1,**4**)}

Priority queue implementations

- ***Array representation (unordered)***. simplest priority queue implementation is based on our code for pushdown stacks.
  - INSERT is the **same** as for *push* in the **stack**.
  - EXTRACT-MAX, we can add code like the inner loop of selection sort to exchange the maximum item with the item at the end and then delete that one, as we did with `pop()` for stacks.
- ***Array representation (ordered).***
  - *insert* as in insertion sort).
  - Thus the largest item is always at the end, and the code for *remove the maximum* in the priority queue is the same as for *pop* in the stack.

Priority queue implementation

Using all the elements in an unsorted list.



23:1 INSERT O(1)

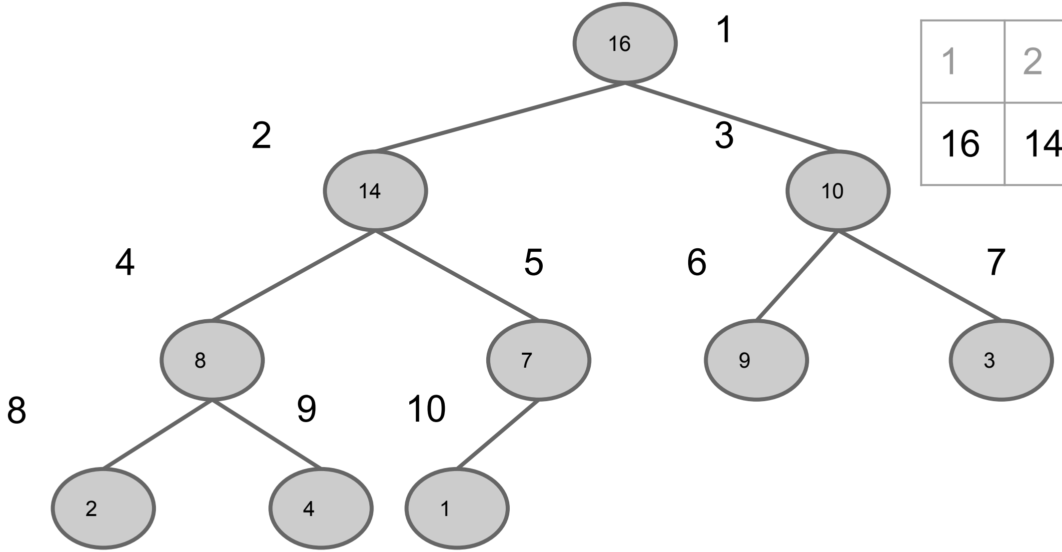
EXTRACT-MAX O(n)

Heap

Implementation of a priority queue

An **array**, visualized as a nearly complete **binary tree**

Max Heap Property: The key of a node is ≥ than the keys of its children



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Heap as a Tree (properties)

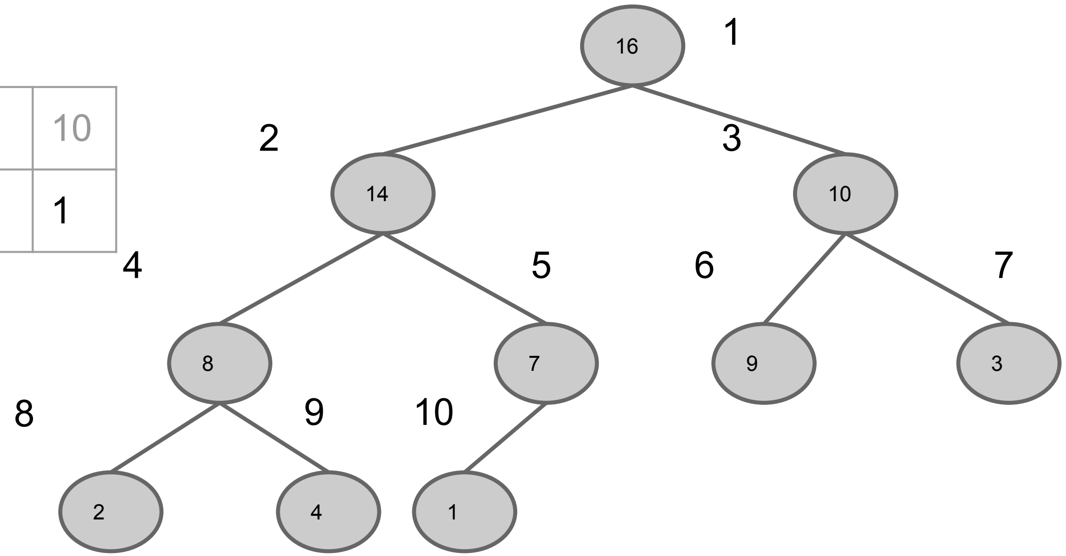
root of tree: first element in the array, corresponding to i = 1

parent(i) =i/2: returns index of node's parent

left(i)=2i: returns index of node's left child

right(i)=2i+1: returns index of node's right child

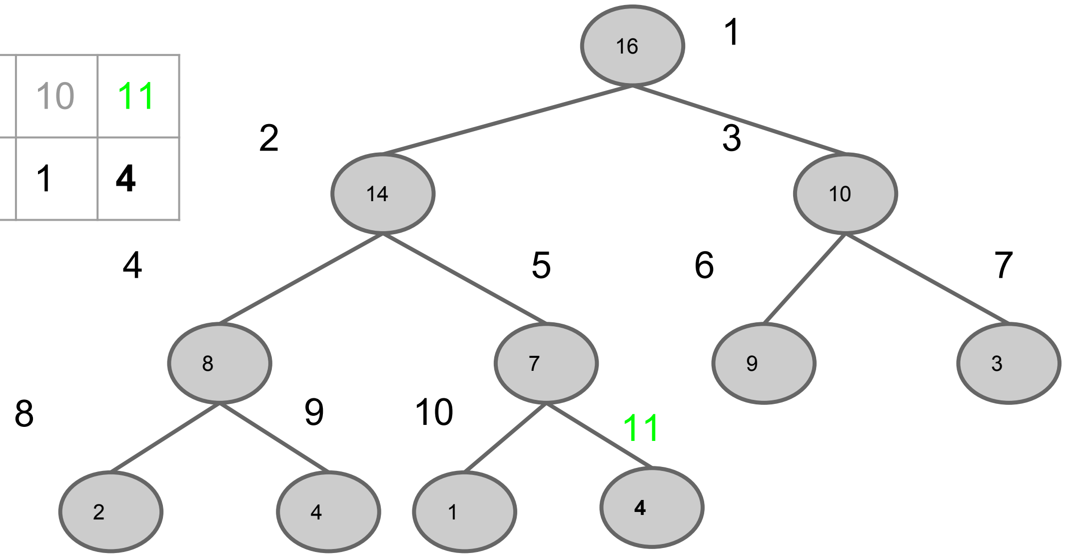
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1



Heap (insert)

Add the element to the bottom level of the heap.

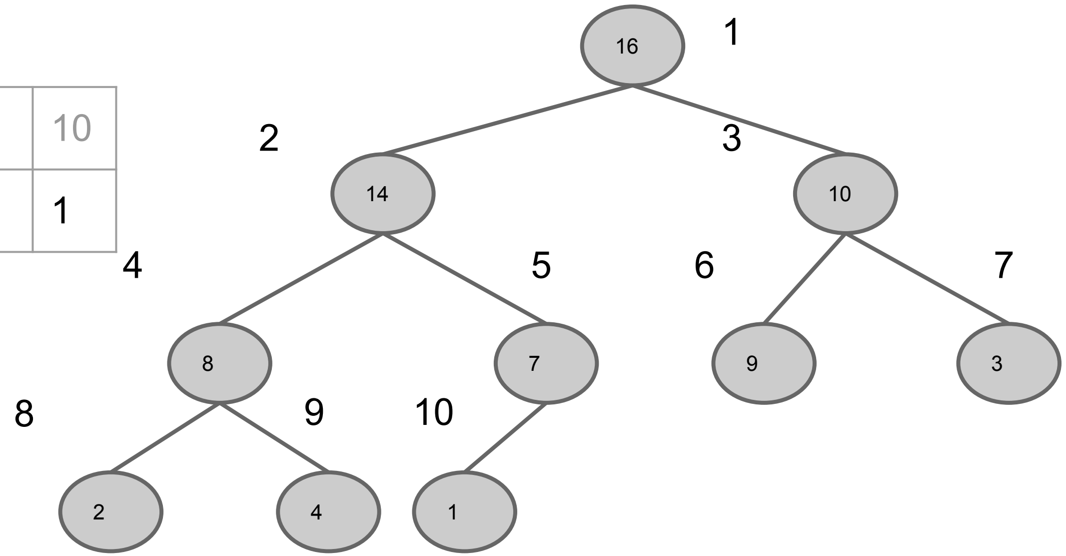
1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	**4**



[Max|Min] Heap Property

- **Max Heap Property:** The key of a node is ≥ than the keys of its children
- (Min **Heap Property** analogously)

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1



More Heap Operations

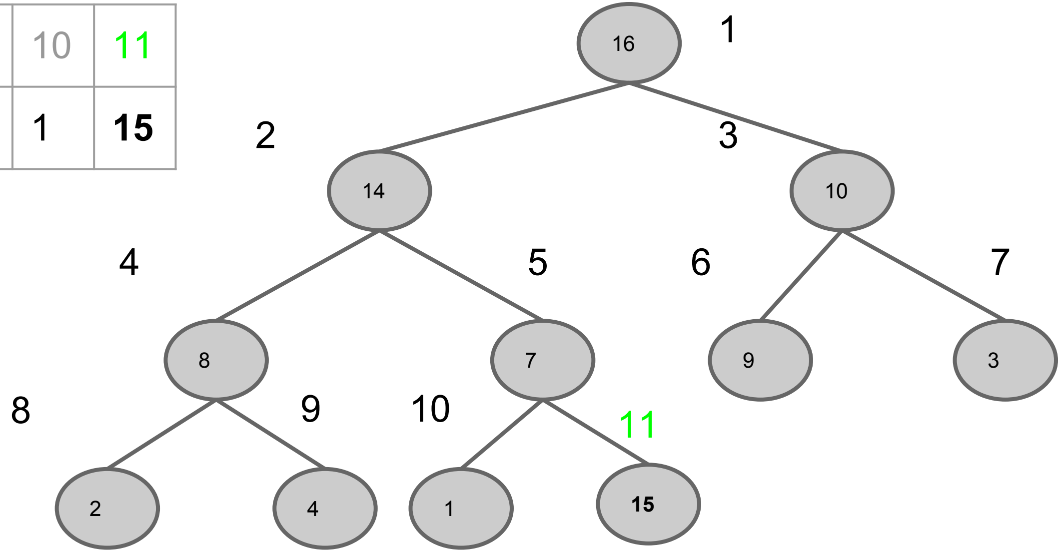
BUILD_MAX_HEAP:produce a max-heap from an unordered array

MAX_HEAPIFY:correct a **single violation of the heap property** in a subtree at its root

INSERT: Insert a element, max/min-heap property is conserved

EXTRACT_MAX:Extract max(or min) a element, max/min-heap property is conserved

1	2	3	4	5	6	7	8	9	10	11
16	14	10	8	7	9	3	2	4	1	**15**



MAX_HEAPIFY

Correct a single violation of the heap property in a subtree

•Assume that the trees rooted at left(i)and right(i)are max-heaps

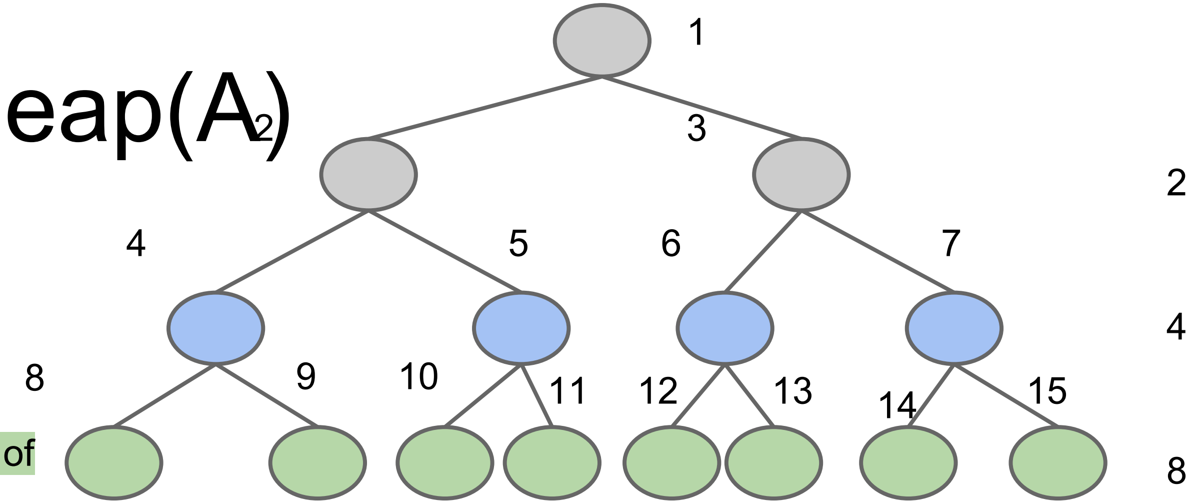
•**If element A[i] violates** the max-heap property, correct violation by **exchanging the appropriate child with the root** and continue r**ecursively down** the tree,making the subtree rooted at index *i* a max-heap.

```
if A[i] has no children
   terminate
x  = max of {A[i],A[i],right(A[i])}
if x = A[i]
   terminate
else
   exchange(A[i] , x)
   max_heapify(x)
```

- Go down and do L levels in the tree, which is logarithmic

O(logn)

Build_Max_Heap(A)



Converts A[1...n] to a max heap

```
Build_Max_Heap(A):
  for i=n/2 to 1
    Max_Heapify(A, i)
```

- A[n/2 + 1 ... n] are all leaves of the tree

- Max_Heapify O(1) for all level above the leaves.

BUILD = O(nLog n) with a more detailed analysis = O(n)

HEAP EXTRACT_MAX

1. Replace the root of the heap with the last element on the last level.
2. **MAX_HEAPIFY(root)**

http://visualgo.net/heap.html

Segment tree

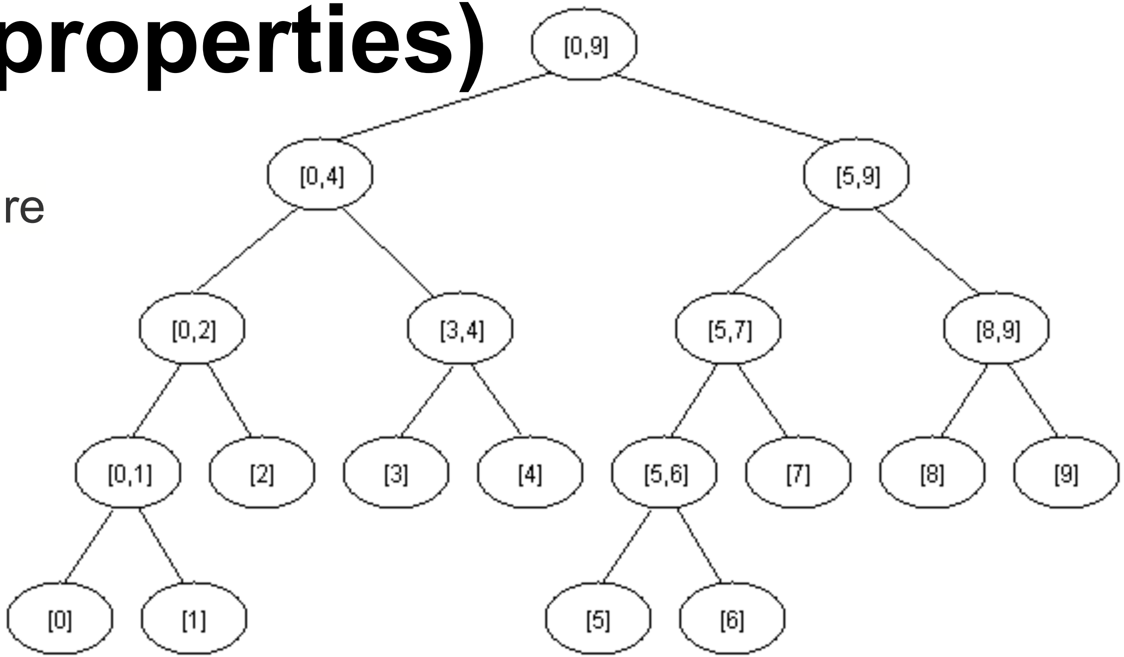
Helps to solve Range [Min|Max|Sum] Query (RMQ)

Range Minimum Query (RMQ) is used on arrays to find the position of an element with the minimum value between two specified indices

0	1	2	3	4	5	6	7

[0 -3] = 0

Segment tree(properties)



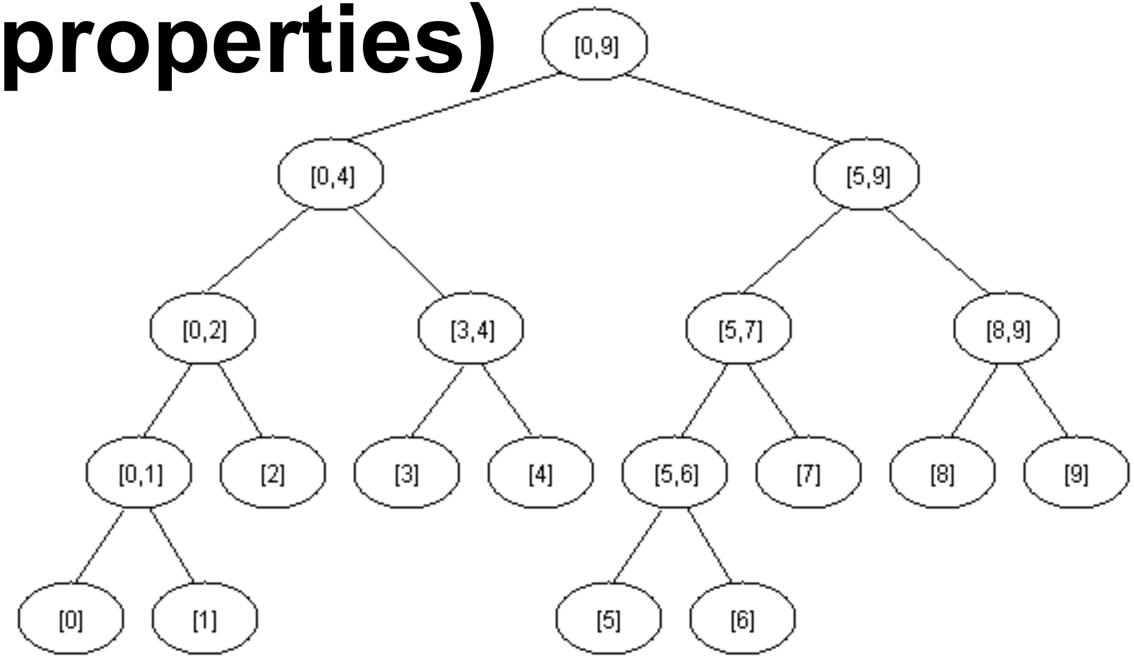
- is a heap-like data structure
- Operations: **update**/query
  - dynamic

- The first node will hold the information for the interval **[i, j]**
  a. the left and right son will hold the information for the intervals **[i, (i+j)/2]** and **[(i+j)/2+1, j]**

Segment tree(properties)

Internal nodes stores the min of a range [i,j]

Array elements at last level



Segment tree(properties)

The segment tree has the same structure as a heap :

A=

0	**1**	**2**	**3**	**4**	**5**	**6**	**7**

A[x] that is not a leaf the :

left(A,i) = **2*x**

right(A,i) =**2*x+1**.

st=

(0,7):min	**(0,3):min**	**(4,7):min**	**(0,1):min**	**(2,3):min**	**...**	**...**	**...**
1	**1*2**	**(1*2)+1**	**(2*2)**	**(2*2)+1**			

Segment tree(build)

```
build(int node, int L_range, int R_range): # O(n log n)
    if (L_range == R_range):          # as L == R, either one is fine
      segment_tree[node] = L;         # store the index
    else :                            # recursively compute the values
      build(left(node) , L              , (L + R) / 2)
      build(right(node)    , (L + R) / 2 + 1,     R       )
      l_node = st[left(node)]
      r_node = st[right(node)];
      segment_tree[node] = (A[p1] <= A[p2]) ? l_node : r_node
```

O(n log n)

Segment tree(Query)

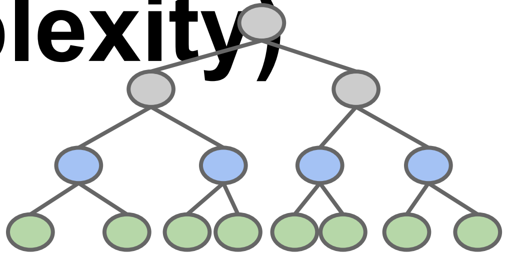
```
int rmq(int node, int L, int R, int i, int j): # O(log n)
    if (i >  R or j <  L) return -1 # Check in  Range in RANGE
    if (L >= i and R <= j) return segment_tree[node]  #  if range
inside query RANGE

    # compute the min position in the left and right part of the
interval
    int l_part = rmq(left(node) , L              , (L+R) / 2, i, j);
    int r_part = rmq(right(node), (L+R) / 2 + 1, R          , i, j);

    if (l_part == -1) return r_part  # if we try to access segment
outside query
    if (r_part == -1) return l_part  # same as above
        return (A[l_part] <= A[r_part]) ? l_part : r_part
```

O(log n)

Segment tree(Update) O(log n)

```
int update_point(int node, int L, int R, int idx, int new_value):
    int i = idx, j = idx;
    # if the current interval does not intersect  # the update interval, return this st node value!
    if (i > R || j < L):
      return segment_tree[node]
    # if the current interval is the index to update,  # update that st[node]
    if (L == i && R == j):
      A[i] = new_value; # update the  array
      return segment_tree[node] = i; // this index
    # compute the minimum pition in the
    # left and right part of the interval
    l_part = update_point(left(node) ,  L              , (L + R) / 2, idx, new_value);
    r_part = update_point(right(node), (L + R) / 2 + 1, R          , idx, new_value);
    # return the position where the overall minimum is
    return st[p] = (A[l_part] <= A[l_part]) ? l_part : r_part
```

Segment tree(Memory Complexity)



Given A[1..n], ST have n leaf.

So, (n-1) internal nodes

Total nodes = 2n-1

Height is: Log2(n)

Total no. of nodes = 2^0 + 2^1 + 2^2 + … + 2^ceil(Log2(n))

by Geometric progression= r* (r^size-1)/(r-1)

= 2*(2^(ceil(Log2(n))+1) -1)/(2-1) = 2* 2 ceil(Log2(n)) -1

Aprox = O(4 * n)

Reference

http://algs4.cs.princeton.edu/24pq/

http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/MIT6_006F11_lec04.pdf

http://www.geeksforgeeks.org/segment-tree-set-1-sum-of-given-range/

Competitive Programming 3: The New Lower Bound of Programming Contests (Steven Halim, Felix Halim)