

AACPP WiSe

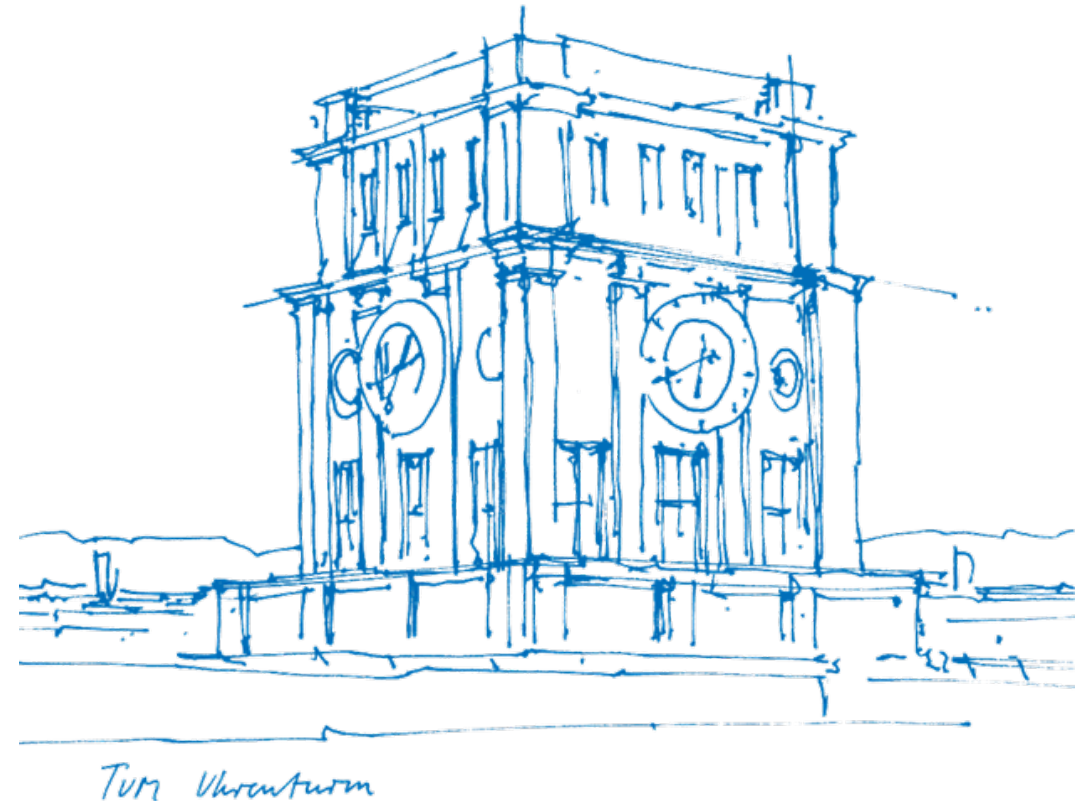
2026/27

Class 1: Binary Search Trees

Mateusz Gienieczko, Julia Spindler

School of Computation, Information and Technology
Technical University of Munich

2026.10.22



Binary Search Trees



General data structure to store keys ordered by some total order, search time $\mathcal{O}(\text{depth})$.

Balanced BST is one where depth is $\mathcal{O}(\log n)$.

AVL and Red-Black Trees are classic.

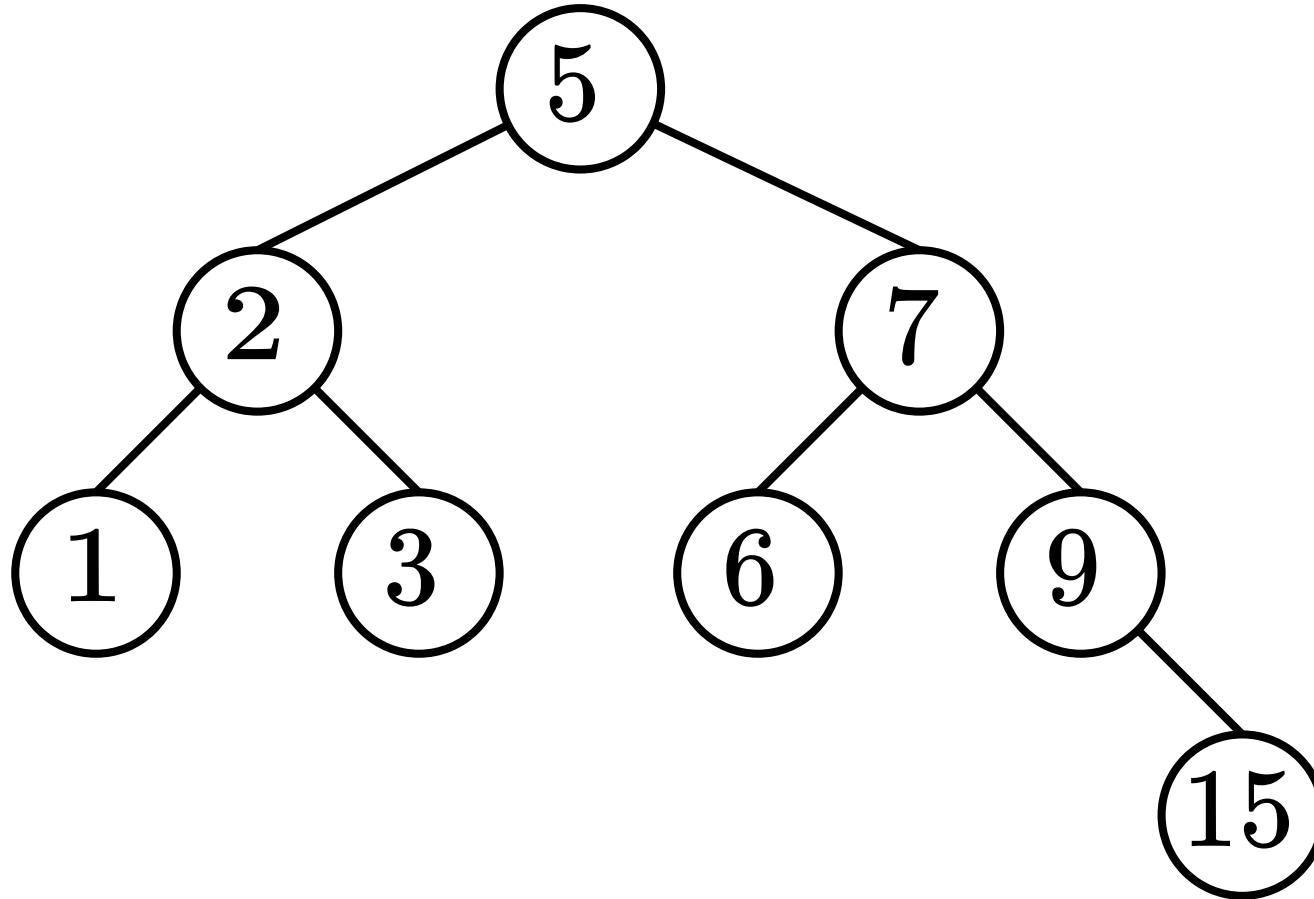
I am assuming everyone signing up for *Advanced* algorithms is aware of the above concepts. We'll cover more interesting BSTs and segment trees.

Splay is a BST that is fast in practice – it tries to keep frequently accessed nodes close to the root.

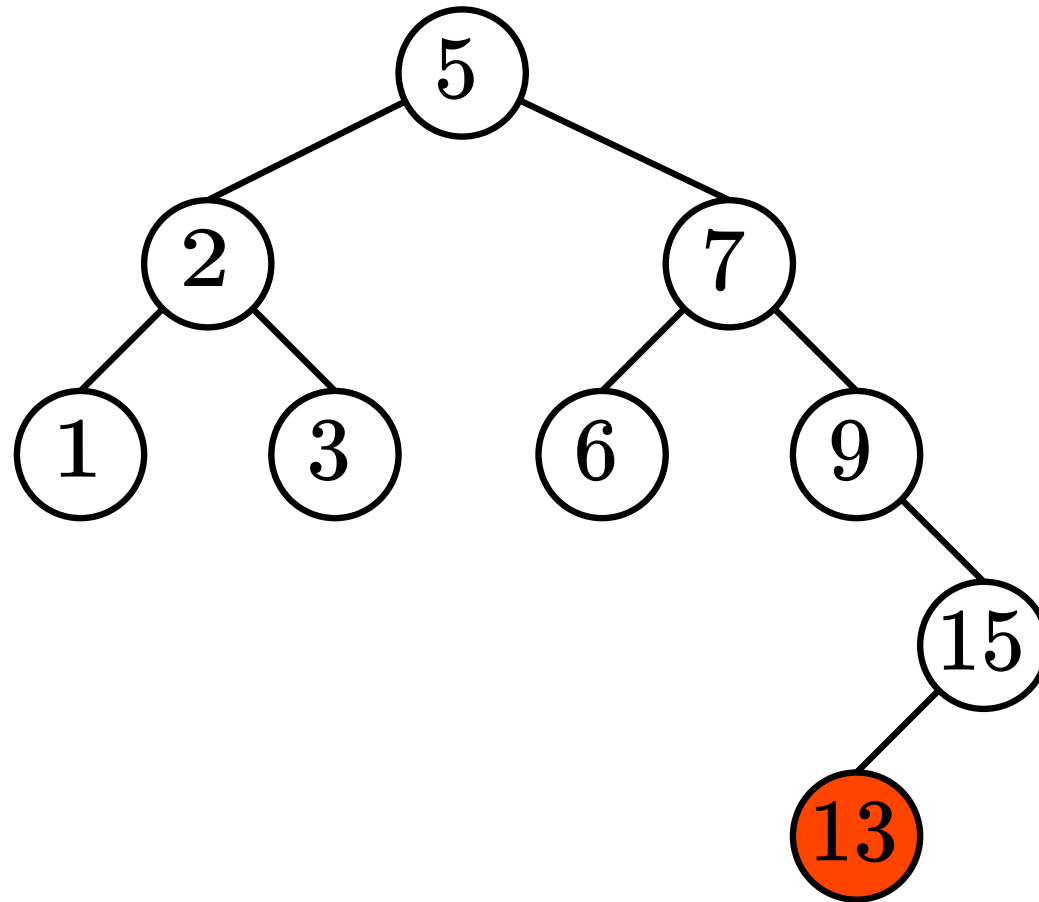
Its runtime is *amortised* $\mathcal{O}(\log n)$ for access and update.

It's based on a splay operation which pushes a node upwards until it becomes the root.

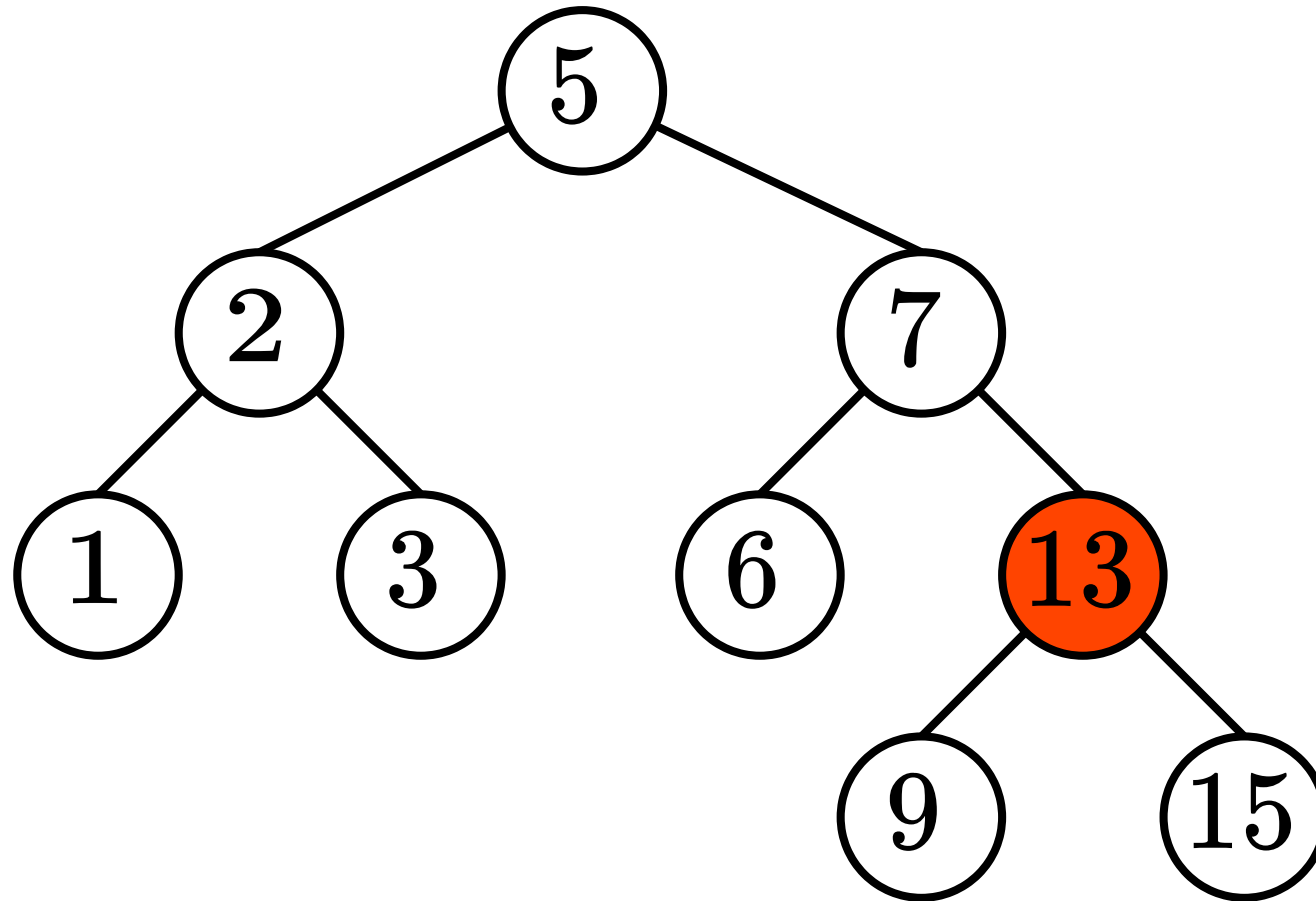
BST – Splay (insertion)



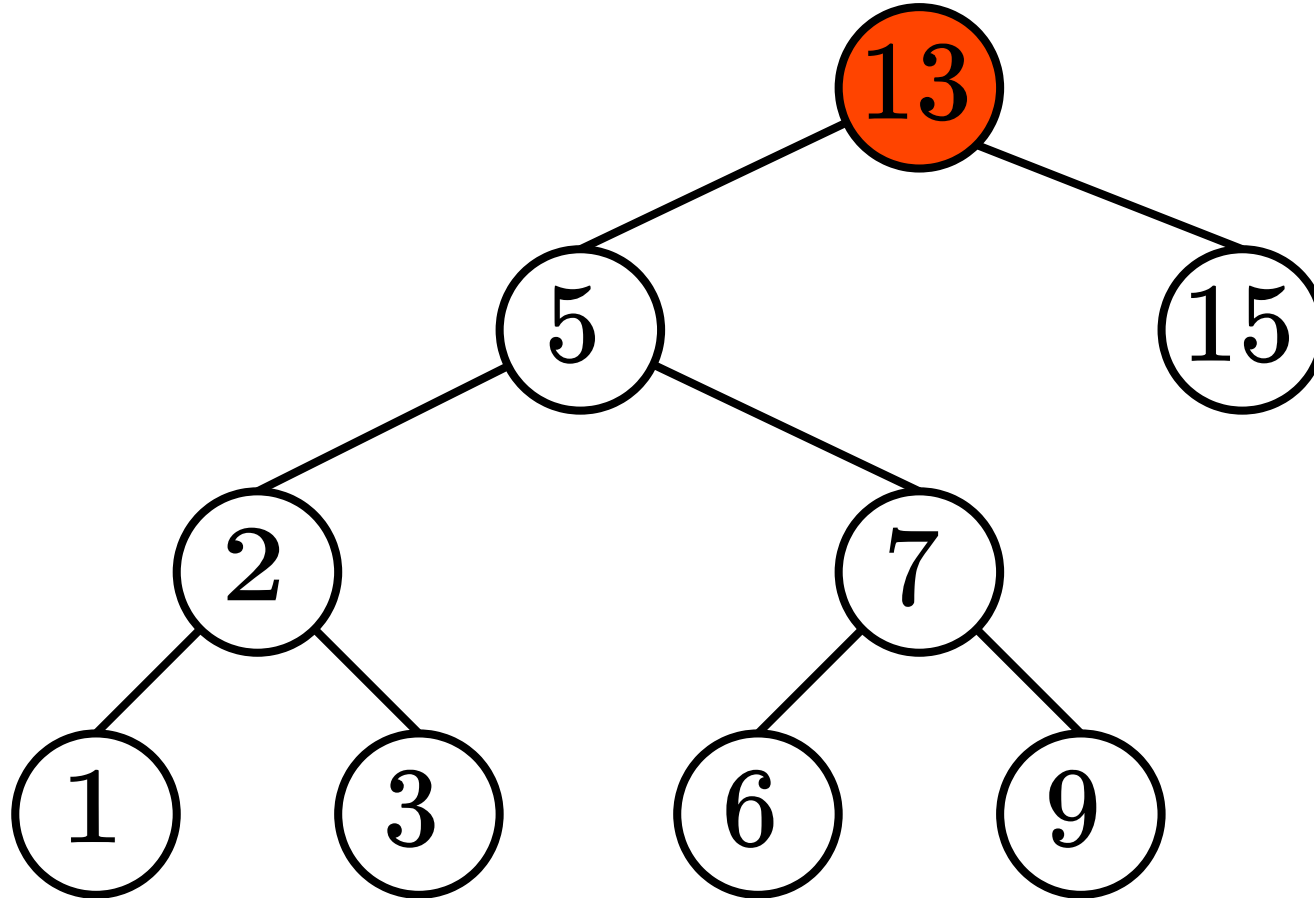
BST – Splay (insertion)



BST – Splay (zag-zig)



BST – Splay (zag-zag)

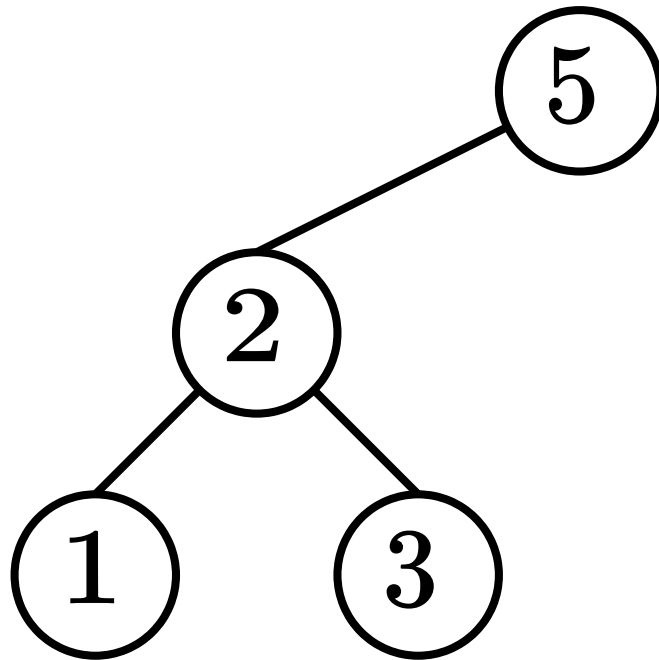
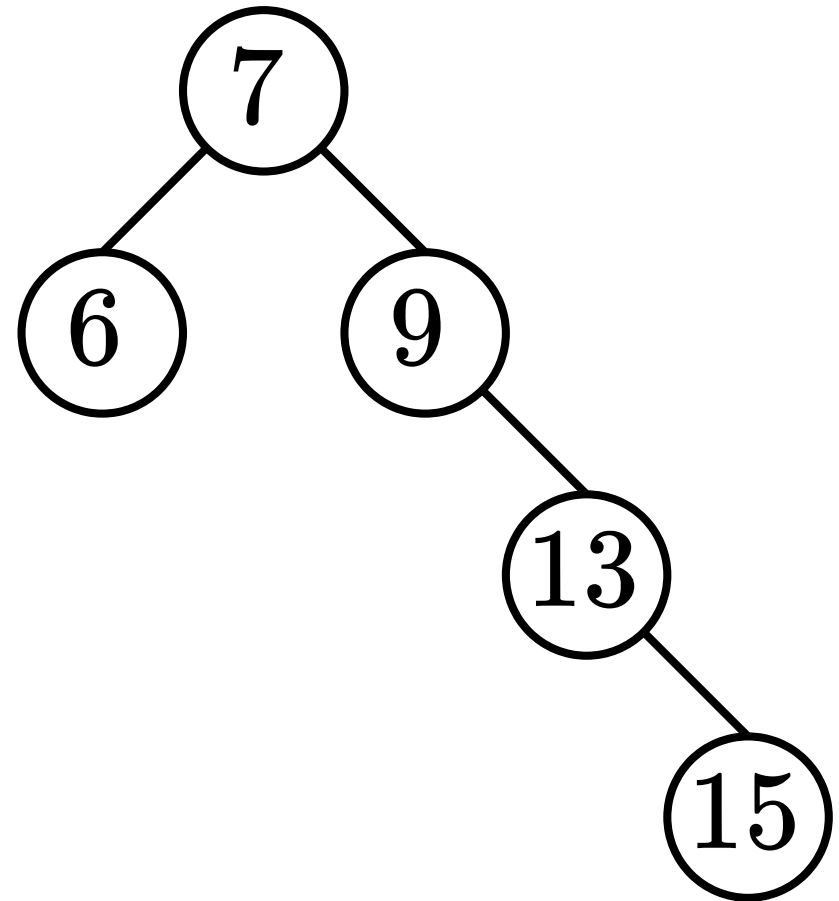


The cool part of splay trees is that we can easily join two splays or split a splay on a given key.

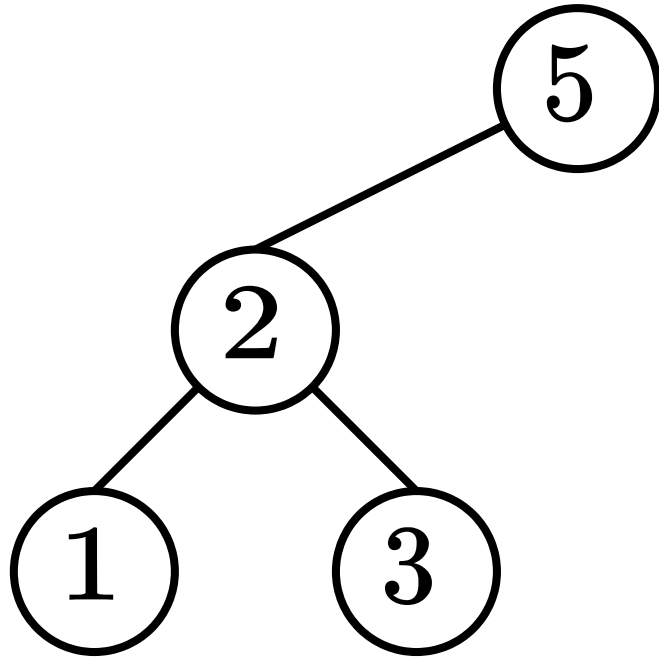
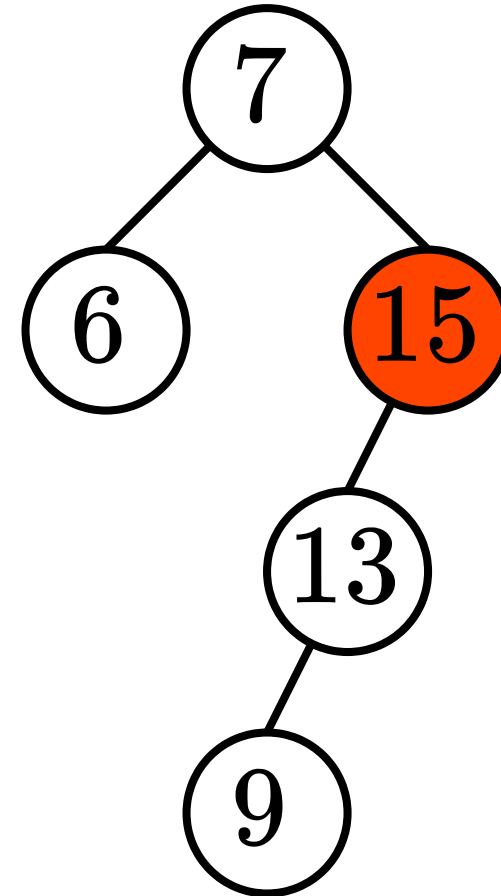
This then allows us to implement all other operations based on join and split.

- Insert – split on the key, create a new node as the root.
- Delete – splay the node to the top, remove it, join its children.

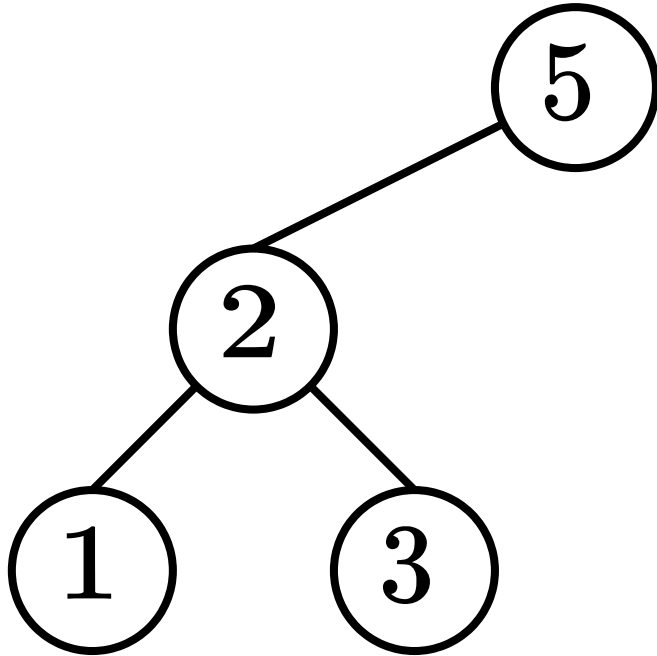
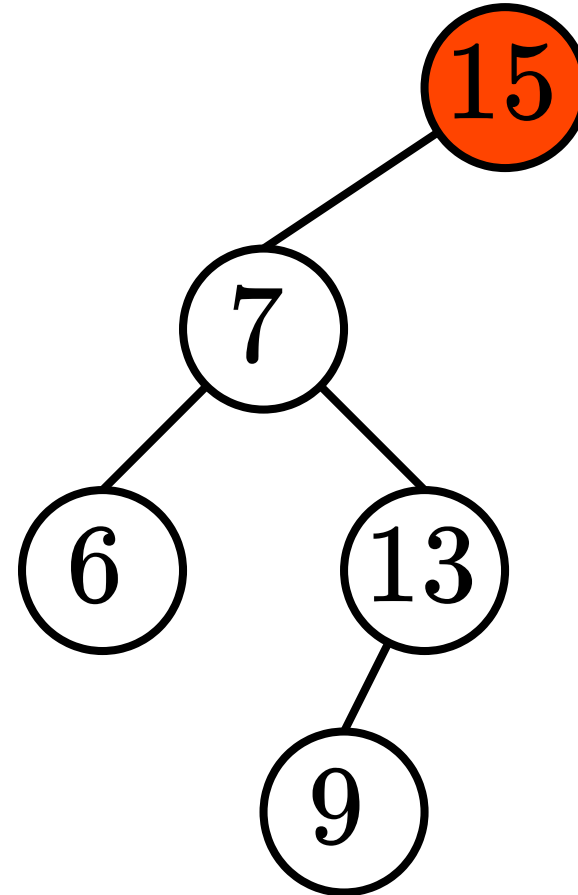
BST – Splay (join)



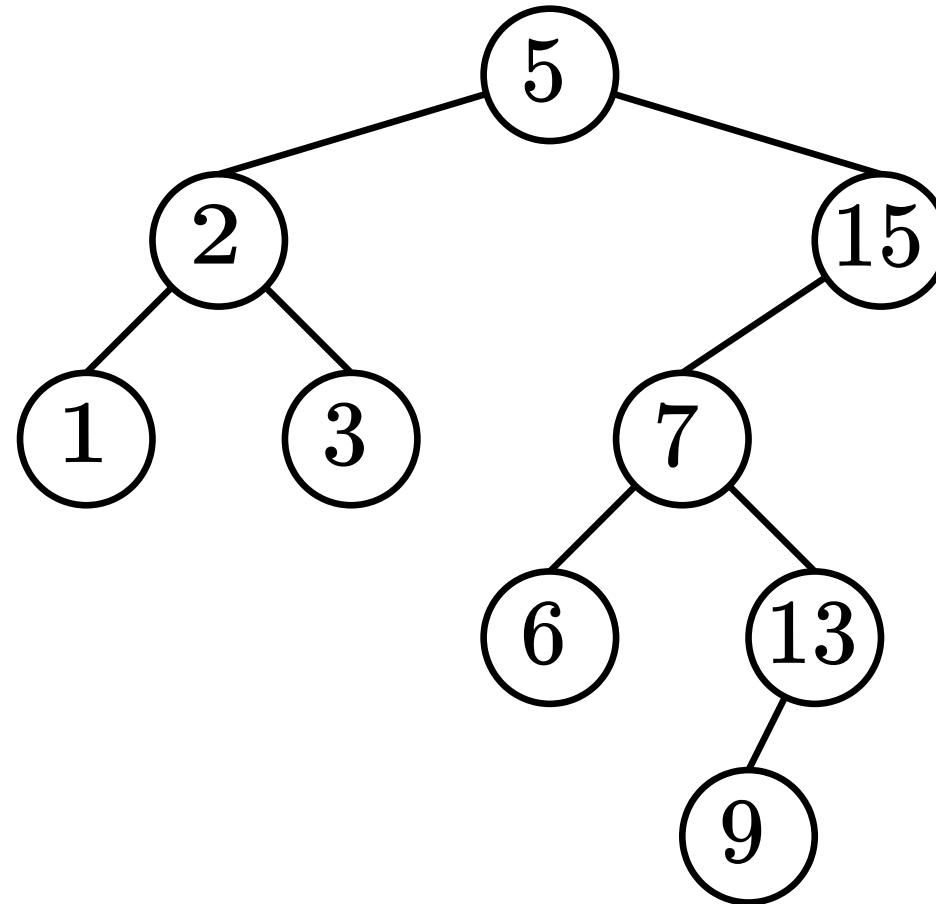
BST – Splay (join)



BST – Splay (join)



BST – Splay (join)



A treap is a hybrid of a tree and a heap.

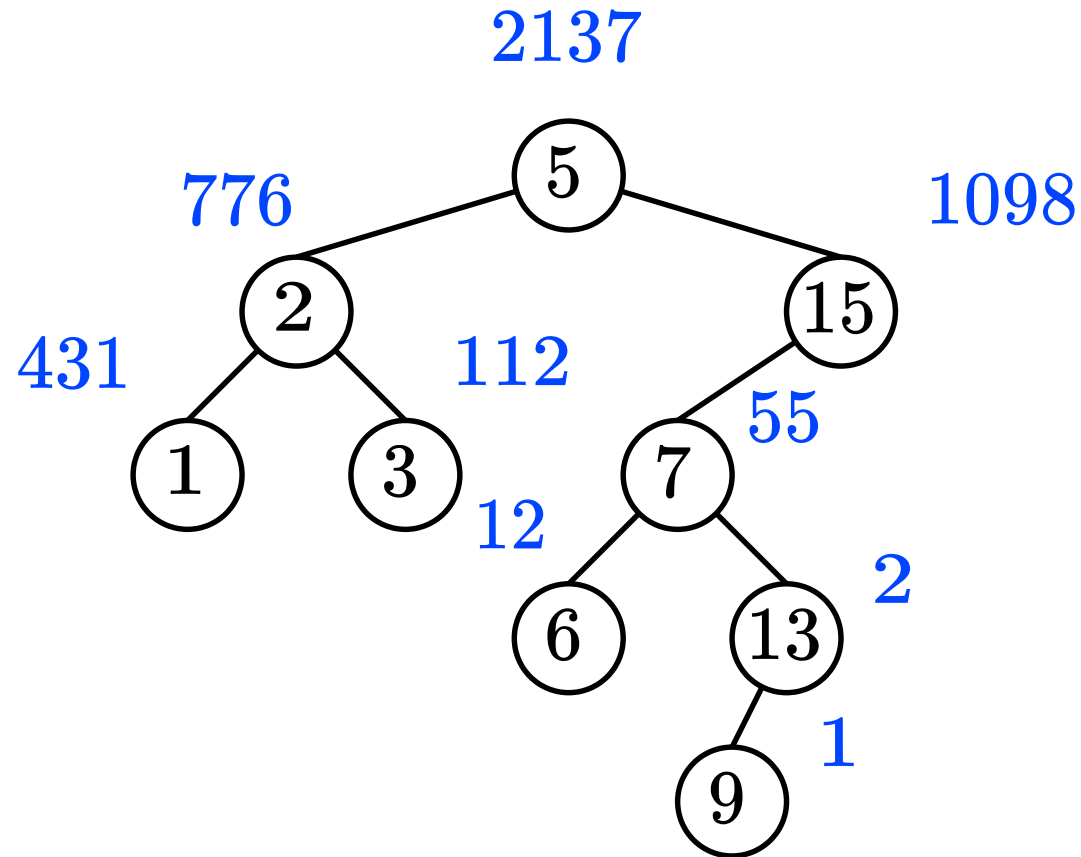
It's based on the following fact:

A random binary tree of n nodes has height $\mathcal{O}(\log n)$.

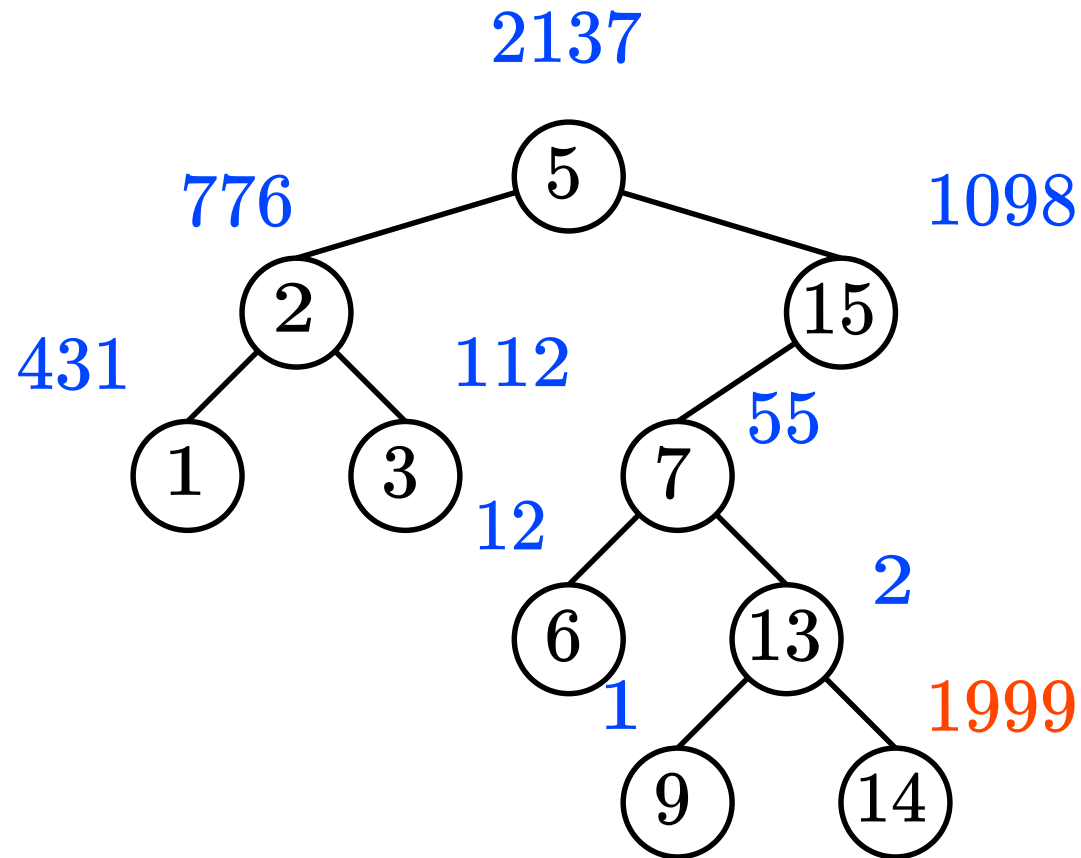
We maintain a BST, but also assign *pseudorandom* weights to each node and maintain the heap invariant (children's weight does not exceed the parent's).

It can be proven that this gives average lookup and insertion of $\mathcal{O}(\log n)$.

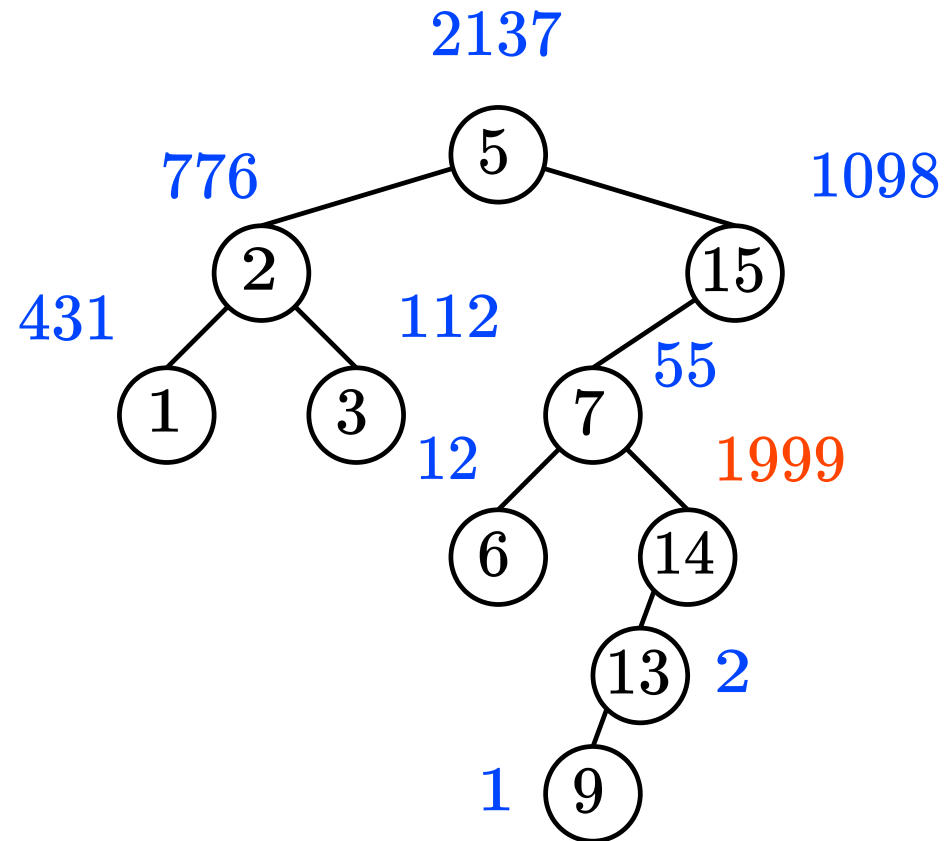
BST – Treap



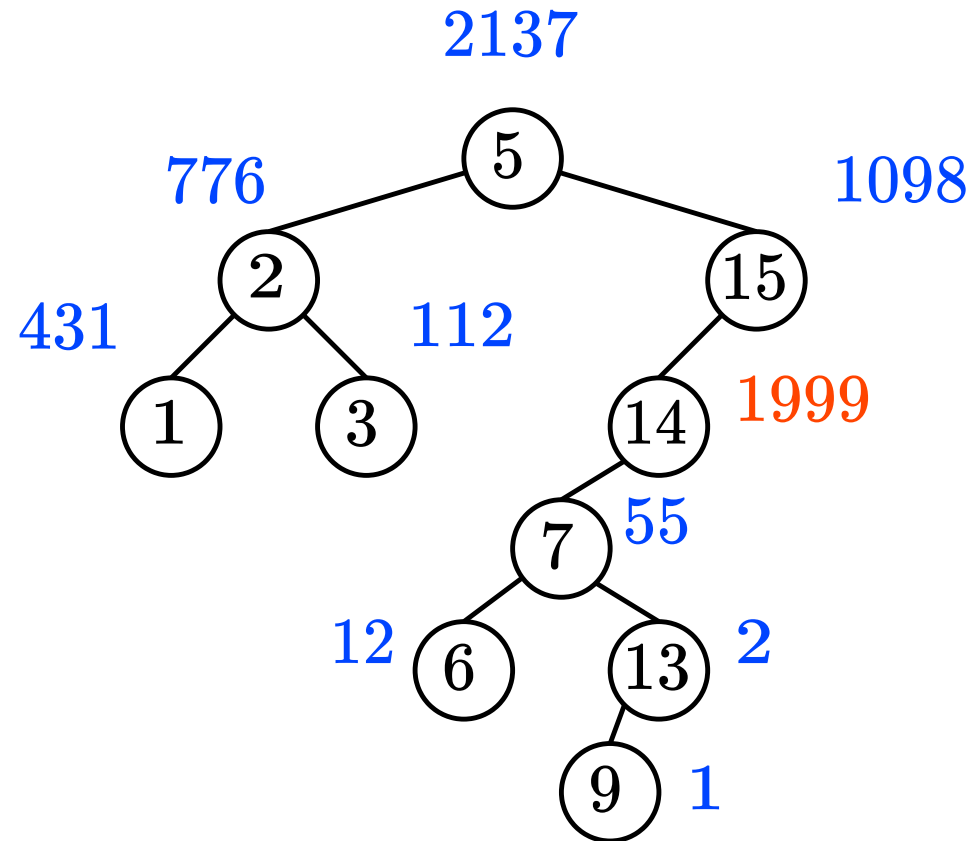
BST – Treap



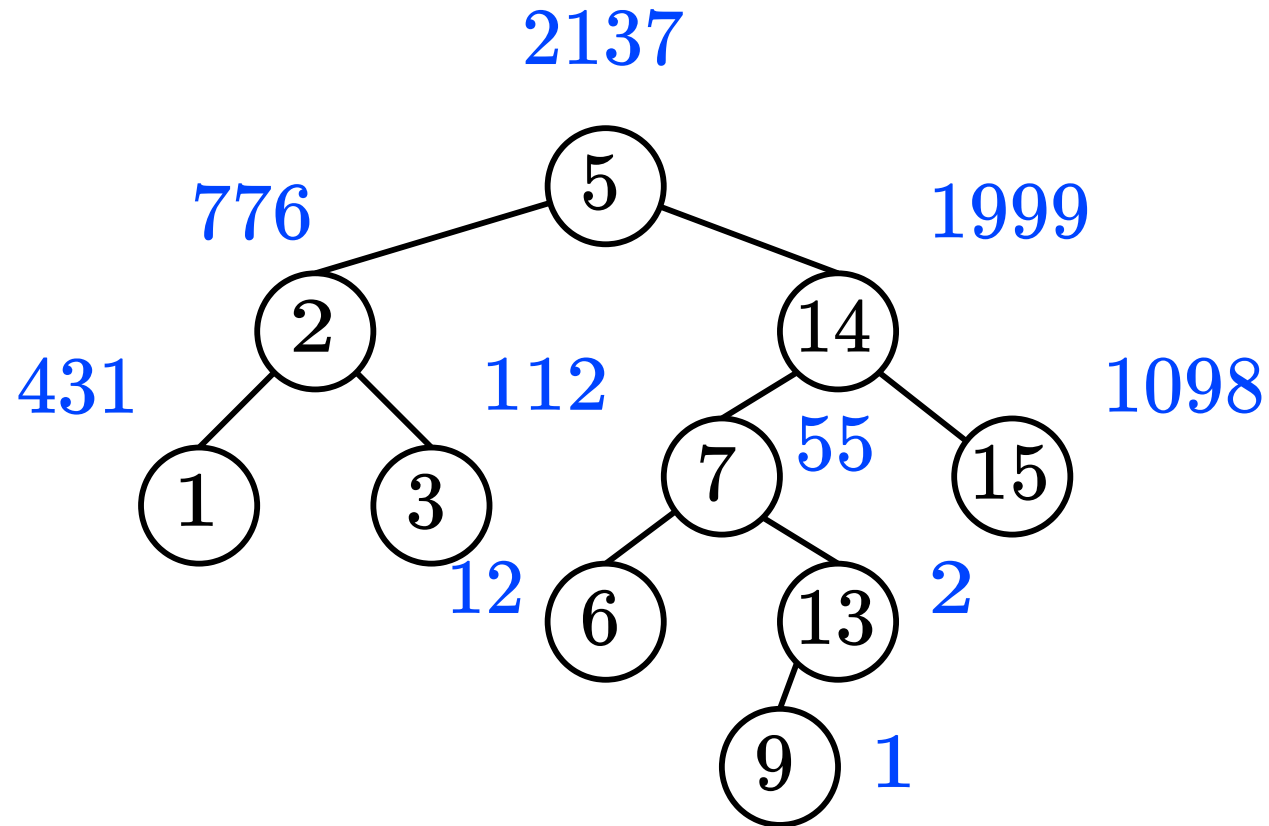
BST – Treap



BST – Treap



BST – Treap



BSTs can be *augmented* with additional information.

We can associate arbitrary values with the keys, creating a map.

Furthermore, we can maintain *aggregates*.

For example, sum of values in a subtree – this allows us to query for the sum of values for all keys in a range $[x, y]$ with two queries in logarithmic time:

$$\text{sum}([x, y]) = \text{sum}([..y]) - \text{sum}([..x))$$

In general, any value that is *associative* can be maintained, i.e.

$$(x \oplus y) \oplus z = x \oplus (y \oplus z)$$

Sums, products, min/max, XOR, etc.

Many BSTs can be implemented using an array, i.e:

- create a static array of n nodes that can be null;
- make the root always node 1;
- the left and right child pointers are indices into the array where the node is.

This doesn't require dynamic allocations, but it also always consumes all the memory for n nodes even if fewer are actually required (good for conpra, questionable in practice).

BST – Implementation notes



The standard way:

```
type NodePtr = Option<Box<Node>>;  
struct Node {  
    left: NodePtr,  
    right: NodePtr,  
    key: K,  
    value: V  
}
```

```
class Node {  
    std::unique_ptr<Node> left;  
    std::unique_ptr<Node> right;  
    K key;  
    V value;  
};
```

A *segment tree* is a search tree specialised for maintaining aggregates on segments.

The root contains aggregates for the segment $[1, n]$.

Children of a node $[x, y]$ contain the aggregates for $\left[x, \left\lfloor \frac{y}{2} \right\rfloor\right]$ on the left and $\left(\left\lfloor \frac{y}{2} \right\rfloor, n\right]$ on the right.

Leaves contain point aggregates $[x, x]$.

We don't need a complex structure for those nodes – a single array of length $2n$ suffices.

Root is 1. The left child of x is $2x$. The right child is $2x + 1$.

Nodes from n to $2n - 1$ are leaves.

Both query and update can be range-based or it can be point-based.

We get three different kinds of trees based on query-update type: point-range, range-point, range-range.

The point-based ones are a bit easier and more lightweight.

BST – Segment trees (point query)

```
fn query(q)
    query_at(q, 1, 1, n)

fn query_at(q, node, n_from, n_to)
    if n_from == n_to
        return tree[node]
    mid = midpoint(n_from, n_to)
    if q <= mid
        return query_at(q, 2 * node, n_from, mid)
    else
        return query_at(q, 2 * node + 1, mid + 1, n_to)
```

BST – Segment trees (point query)

```
fn query(q)
    query_at(q, 1, 1, n)

fn query_at(q, node, n_from, n_to)
    if n_from == n_to
        return tree[node]
    mid = midpoint(n_from, n_to)
    if q <= mid
        return query_at(q, 2 * node, n_from, mid)
    else
        return query_at(q, 2 * node + 1, mid + 1, n_to)
```

Point update is analogous.

BST – Segment trees (range query)

```
fn query_at(q_from, q_to, node, n_from, n_to)
    if q_from == n_from and q_to == n_to
        return tree[node]
    mid = midpoint(n_from, n_to)
    agg = 0
    if q_from <= mid
        agg ⊕= query_at(q_from, min(q_to, mid), 2 * node, n_from, mid)
    if q_to > mid
        agg ⊕= query_at(max(q_to, mid), q_to, 2 * node + 1, mid + 1, n_to)
    return agg
```

To facilitate range-range trees we need *pushdown values*.

The problem is that we need to apply the update on all nodes in the range, but a query might not overlap exactly with the nodes of the update.

E.g an update on $[1, n]$ would touch only the root normally, while a query $[2, 2]$ aggregates only from one node.

Instead, we lazily apply values in top nodes and then *push down* whenever we touch them.

This can be applied to normal BSTs as well (see e.g. RAI from last year).

BST – Pushdown values (range update)



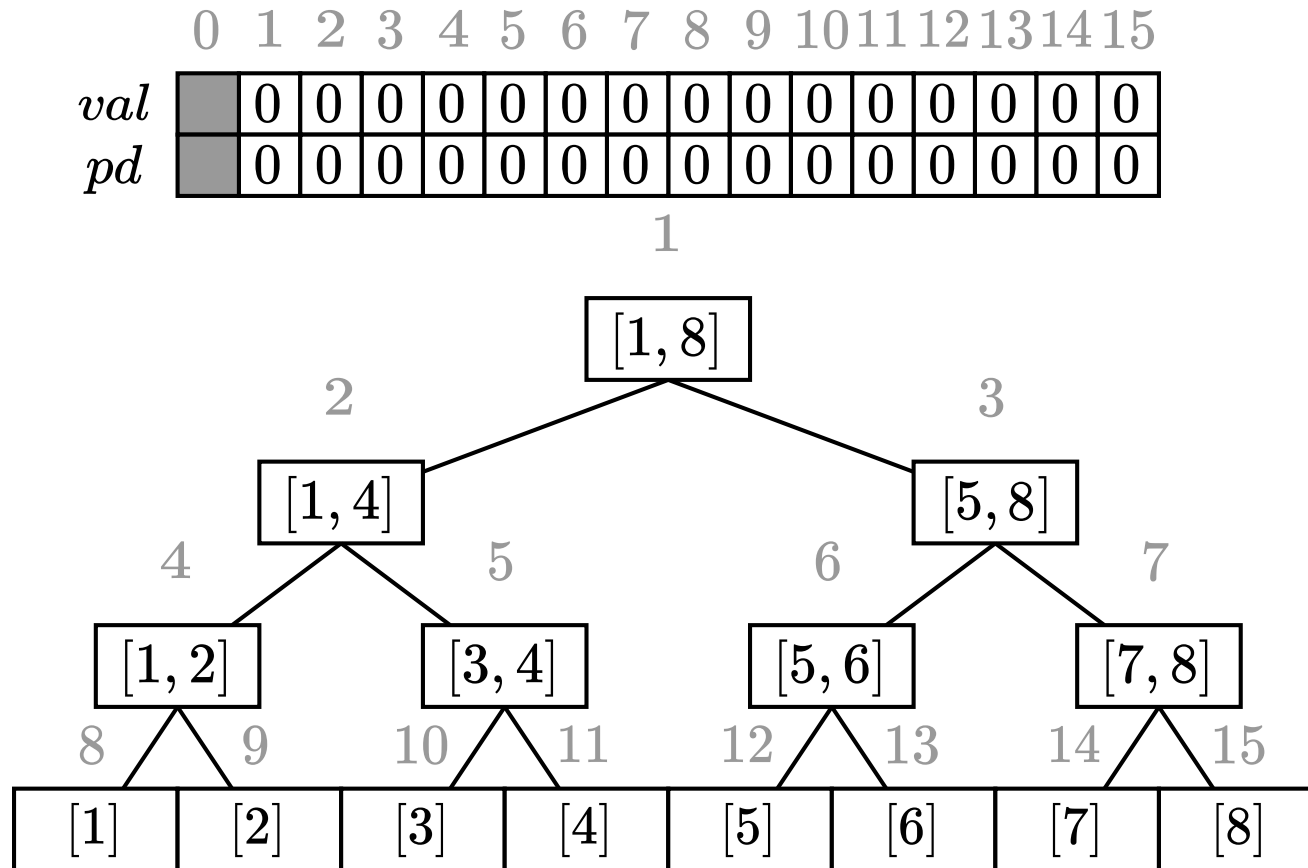
```
fn update_at(q_from, q_to, node, n_from, n_to, value)
    if q_from == n_from and q_to == n_to
        pushdown[node]  $\oplus$ = value
        return
    mid = midpoint(n_from, n_to)
    if n_from <= mid
        update_at(q_from, min(q_to, mid), 2 * node, n_from, mid, value)
    if n_to > mid
        update_at(max(q_to, mid), q_to, 2 * node + 1, mid + 1, n_to, value)
```

BST – Pushdown values (range query)

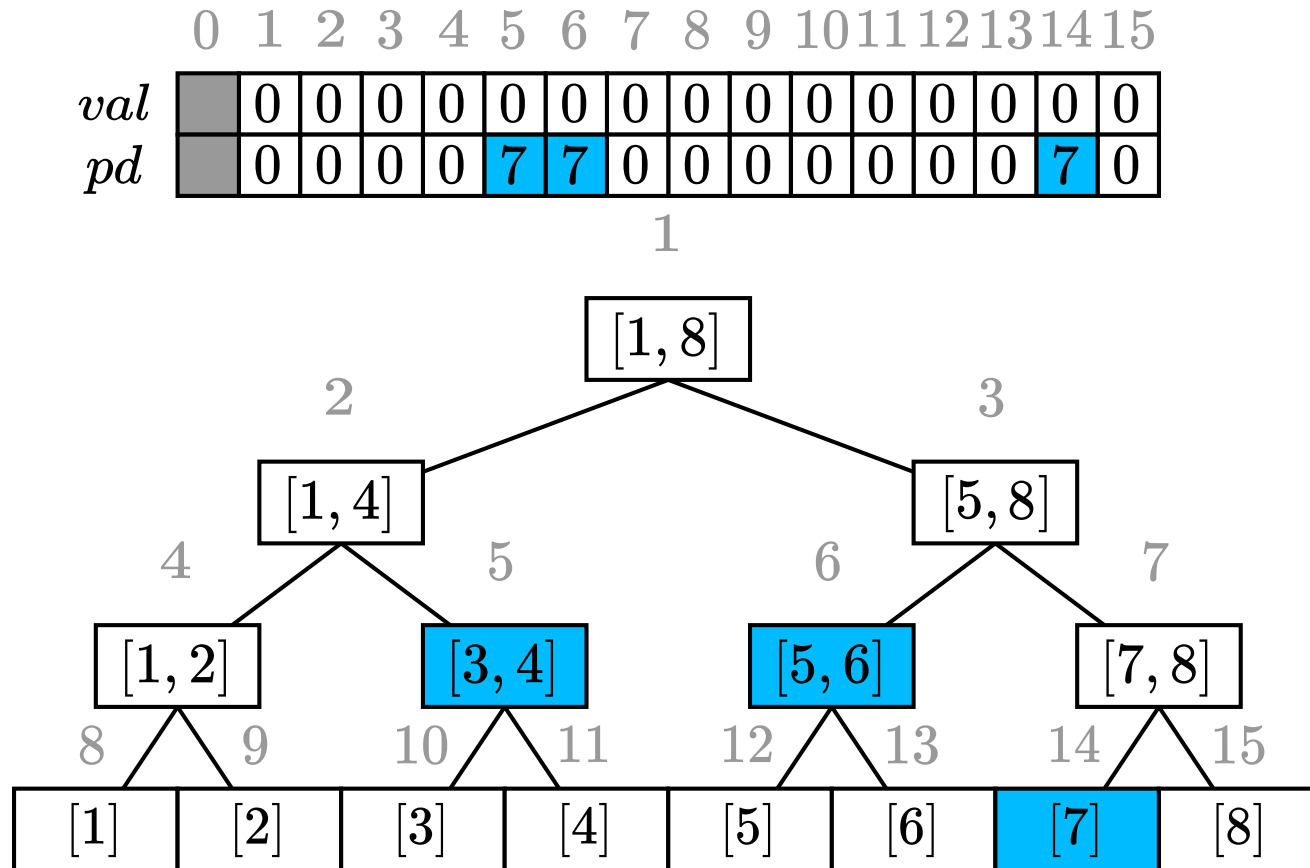
```
fn push_down(node, n_from, n_to)
    tree[node]  $\oplus$ = pushdown[node] * (n_to - n_from + 1)
    if n_from != n_to
        pushdown[2 * node]  $\oplus$ = pushdown[node]
        pushdown[2 * node + 1]  $\oplus$ = pushdown[node]
    pushdown[node] = 0

fn query_at(q_from, q_to, node, n_from, n_to, value)
    push_down(node, n_from, n_to)
    // rest of the code the same as in normal range query
```

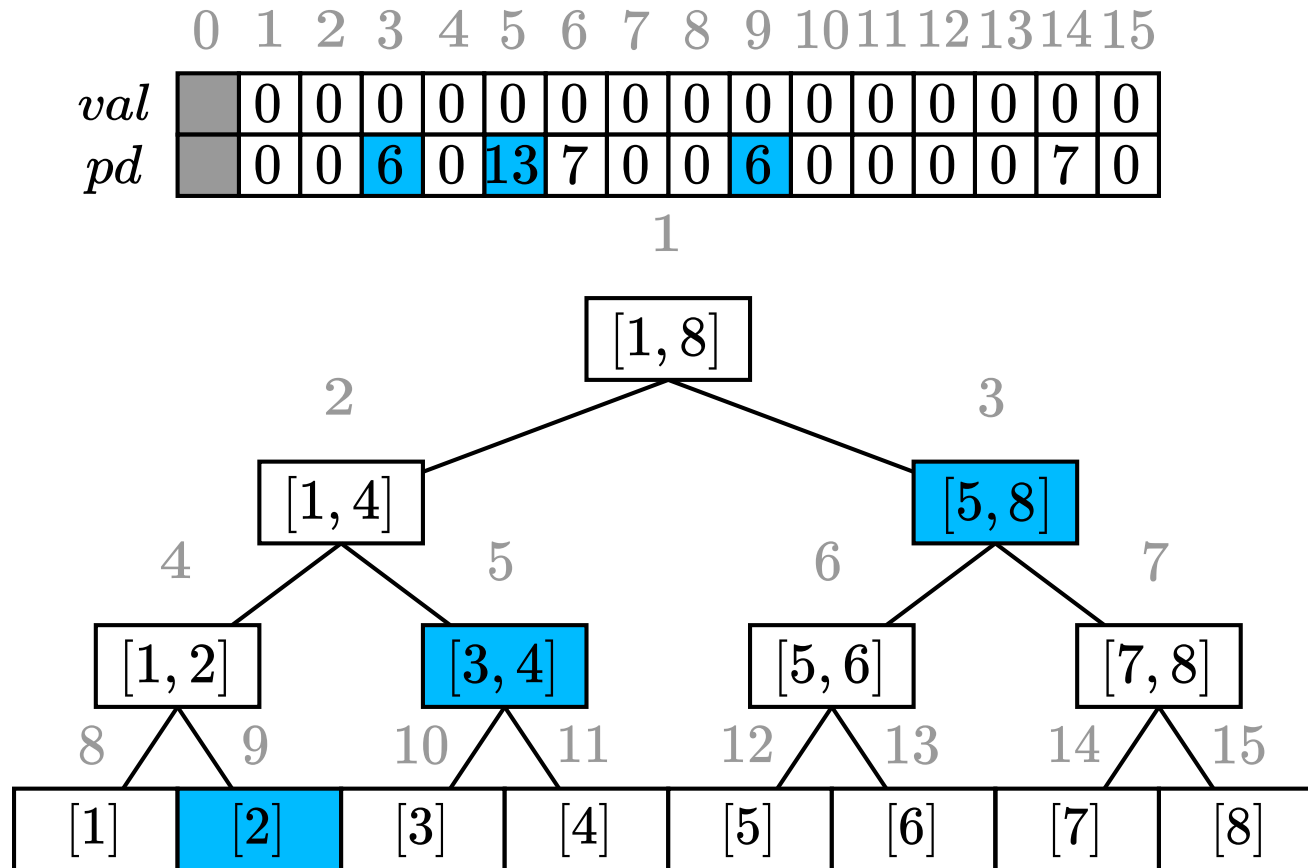
BST – Segment trees



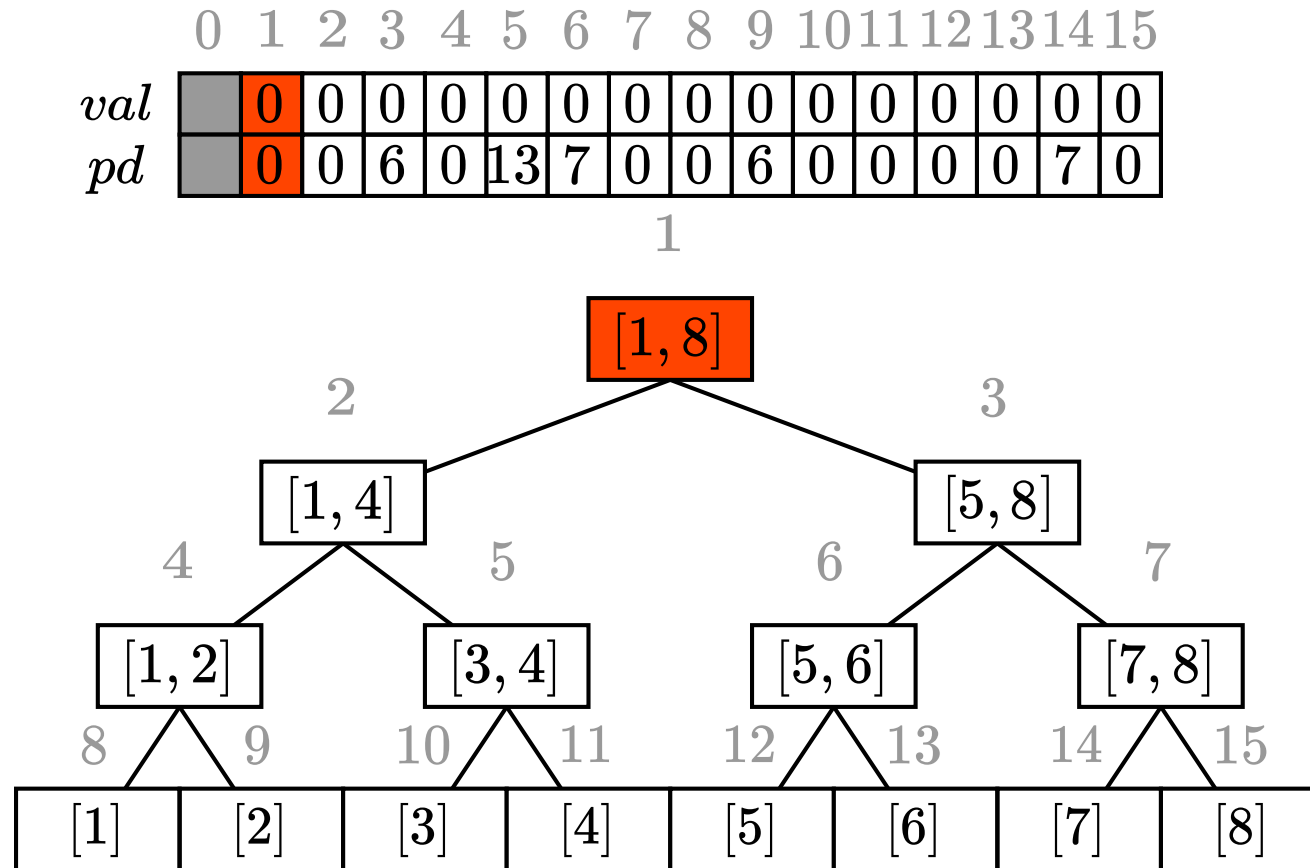
BST – Segment trees (update $[3, 7] + 7$)



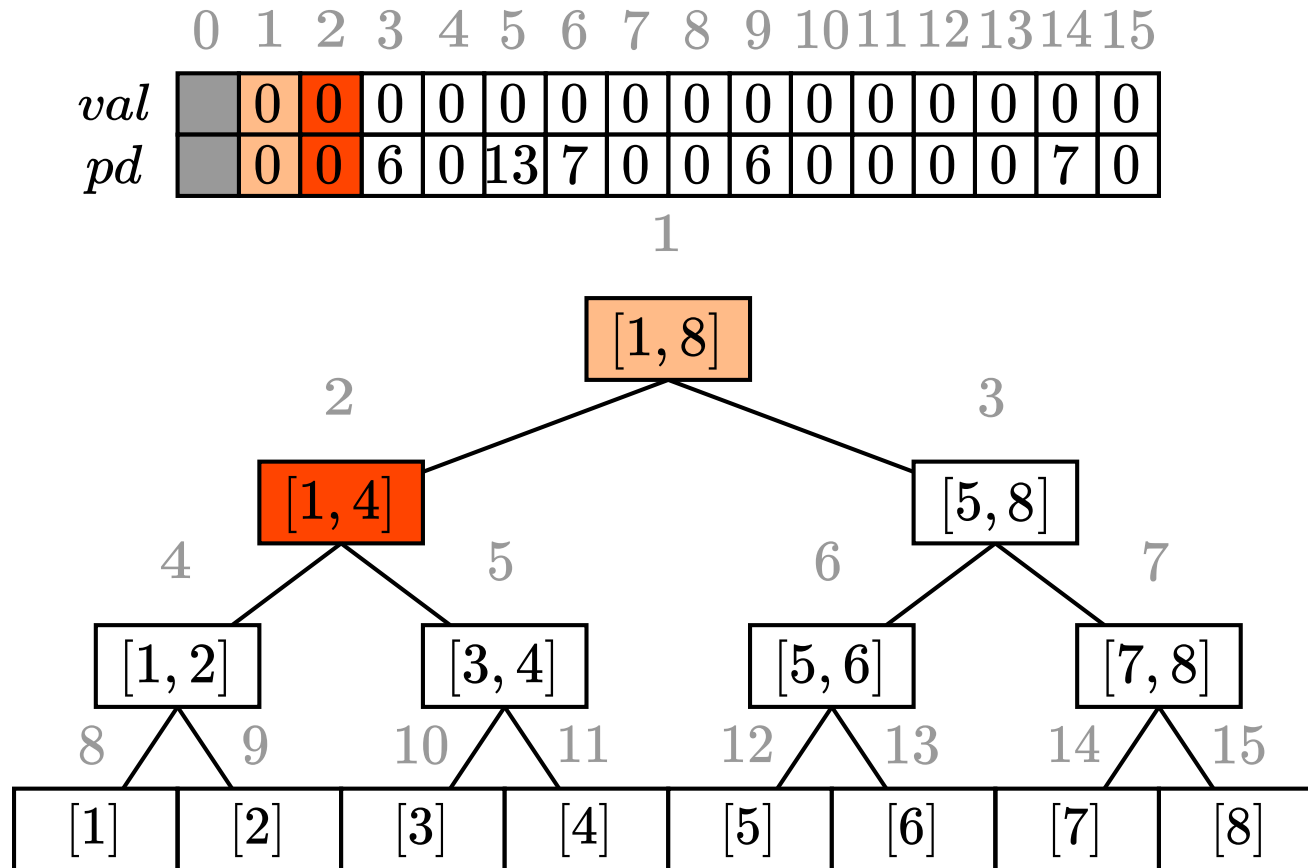
BST – Segment trees (update $[2, 16] + 6$)



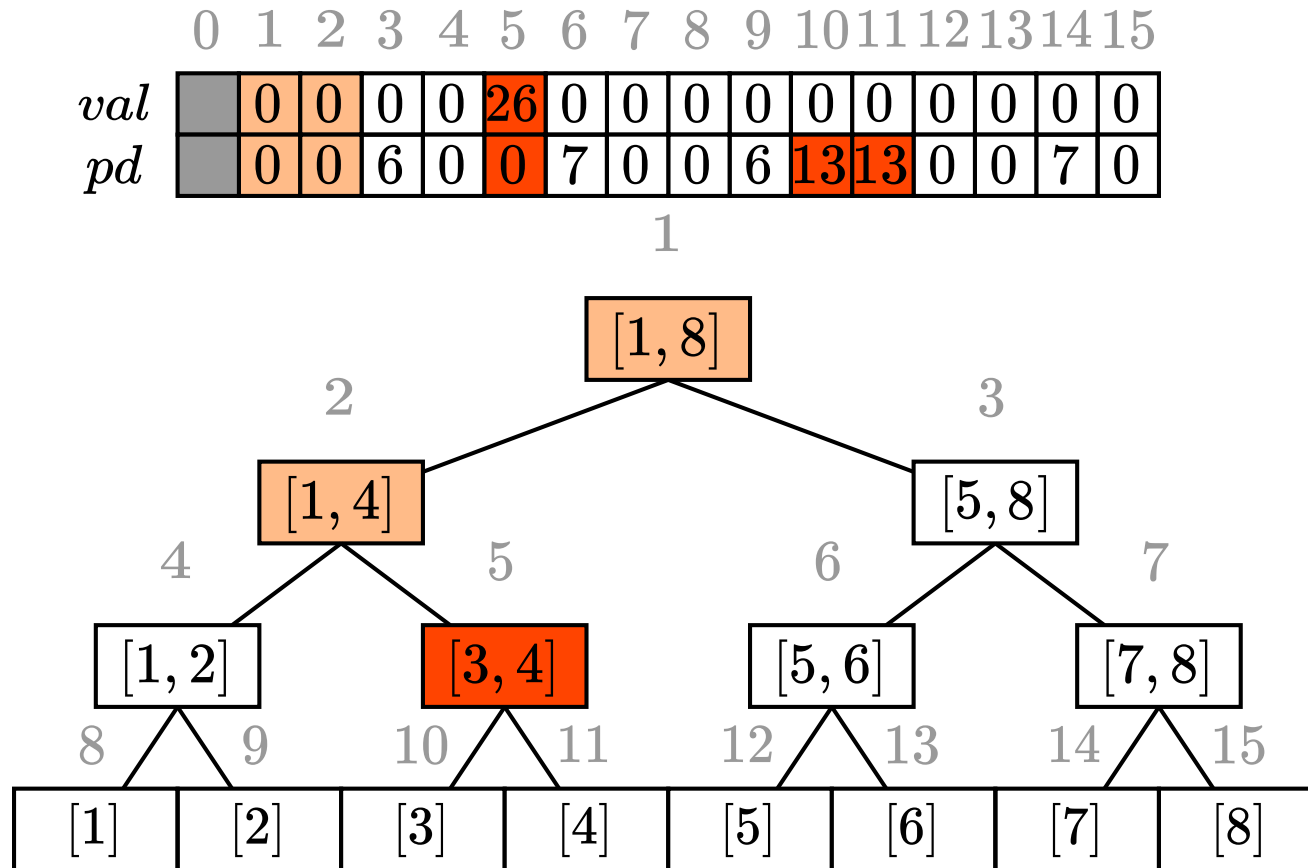
BST – Segment trees (query $\Sigma[4, 6]$)



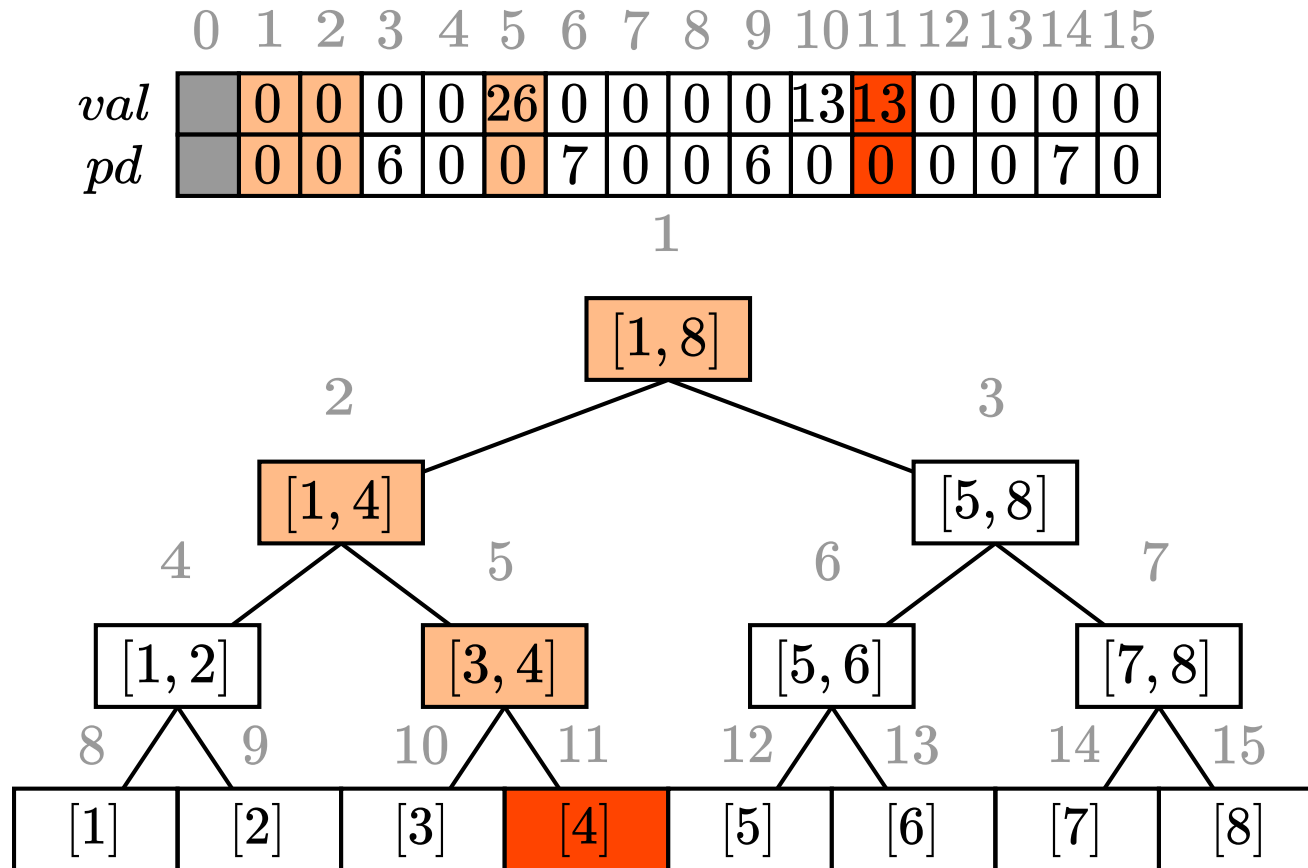
BST – Segment trees (query $\Sigma[4, 6]$)



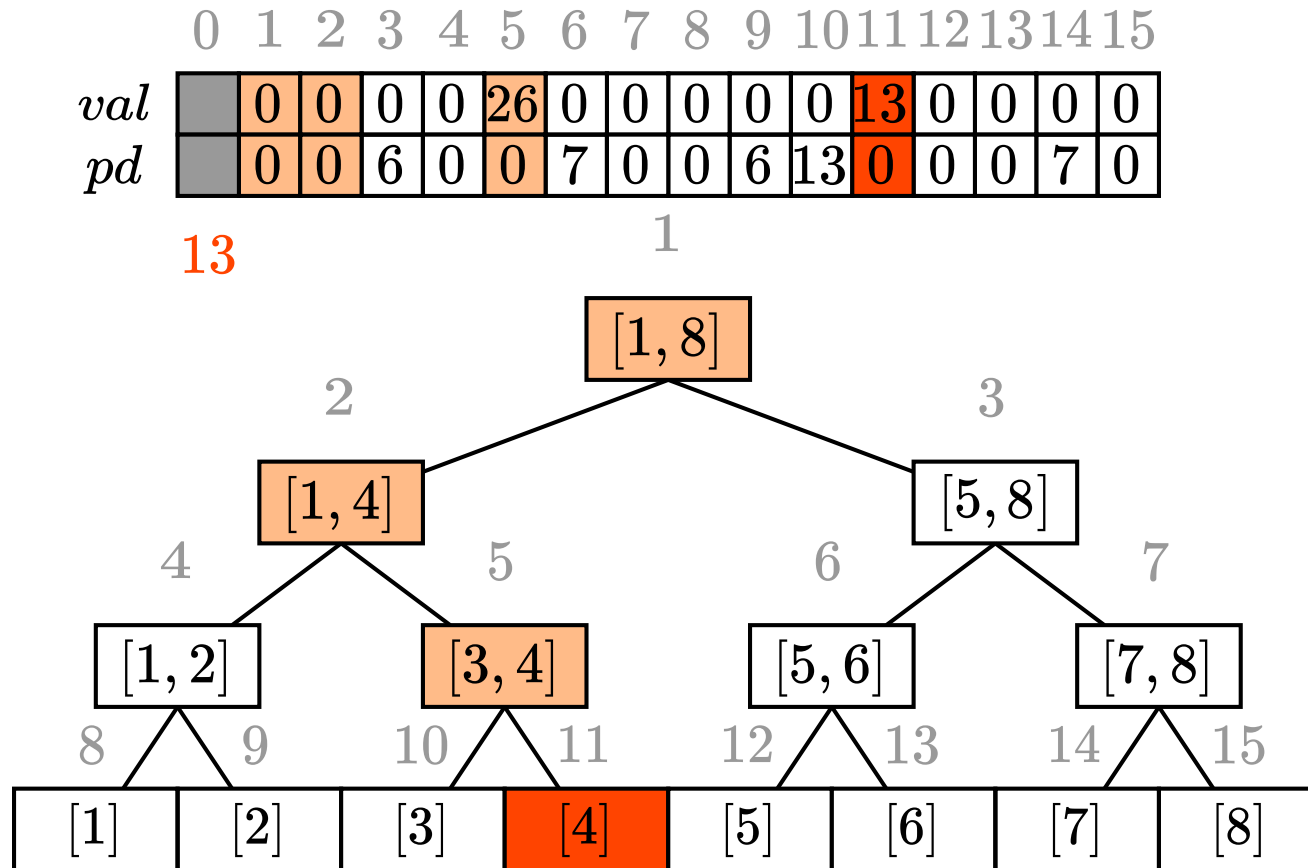
BST – Segment trees (query $\Sigma[4, 6]$)



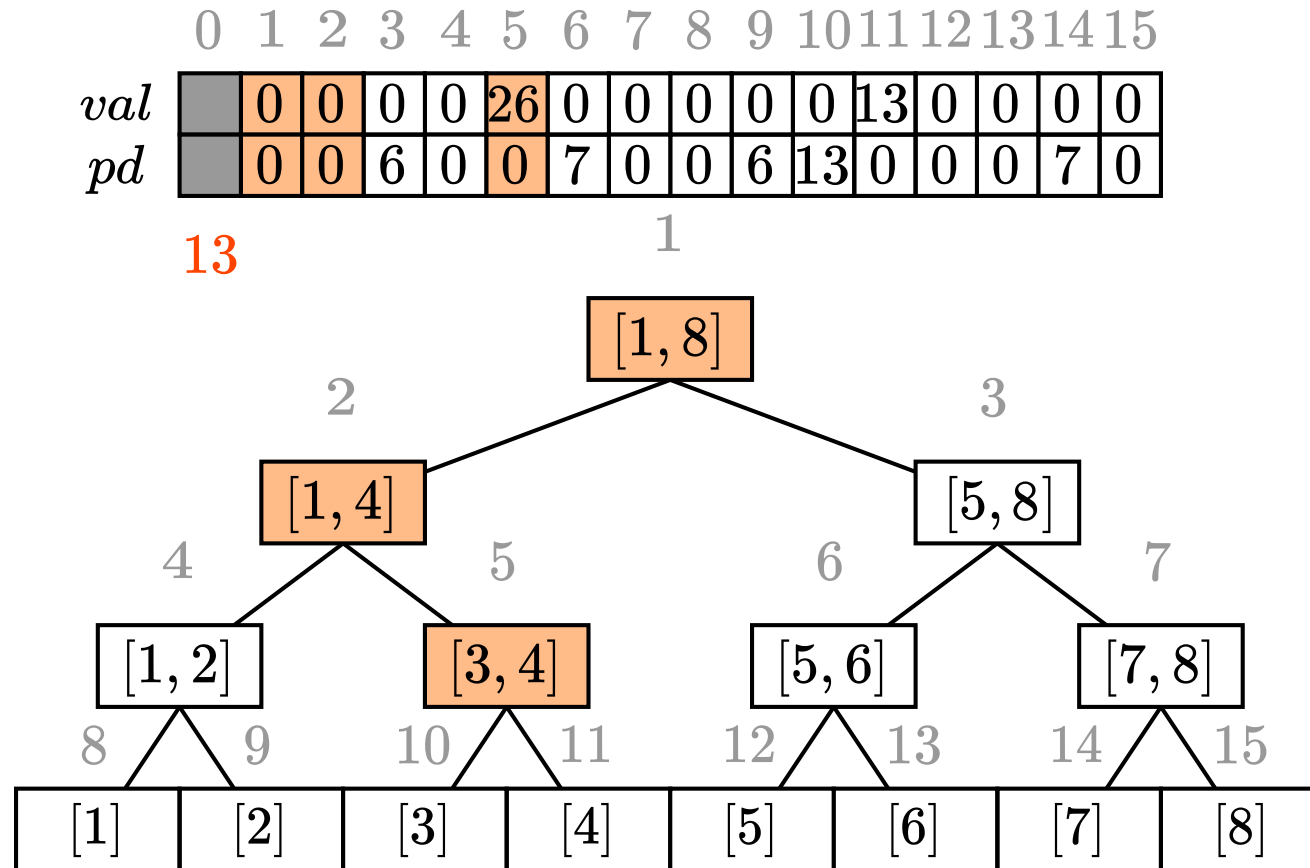
BST – Segment trees (query $\Sigma[4, 6]$)



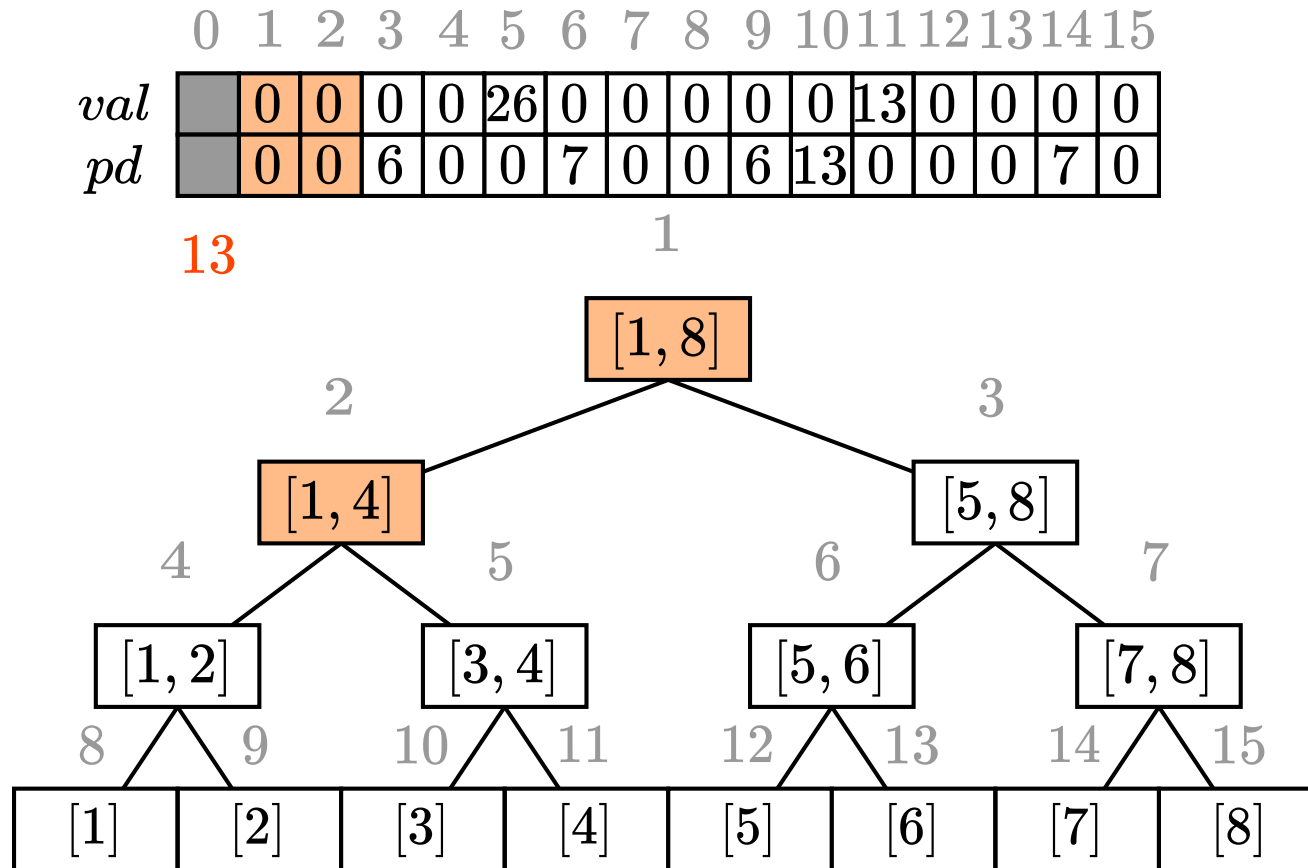
BST – Segment trees (query $\Sigma[4, 6]$)



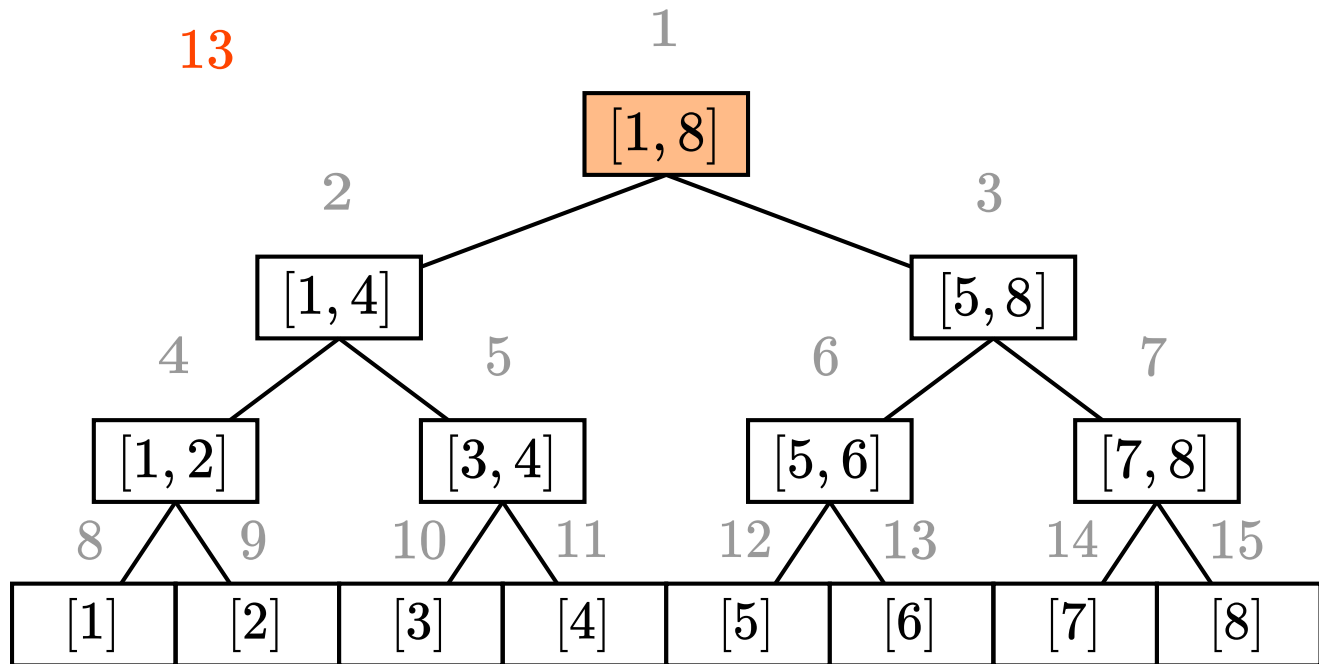
BST – Segment trees (query $\Sigma[4, 6]$)



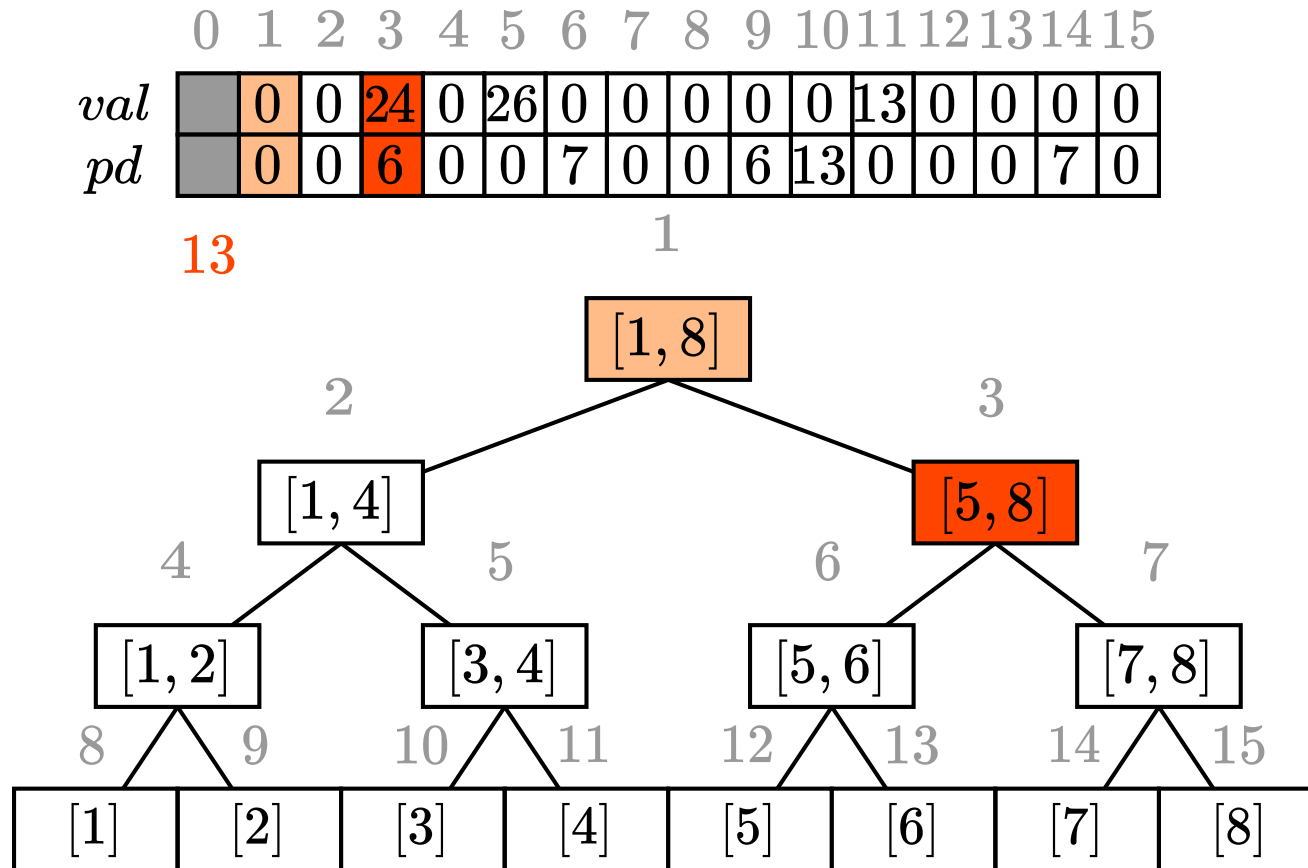
BST – Segment trees (query $\Sigma[4, 6]$)



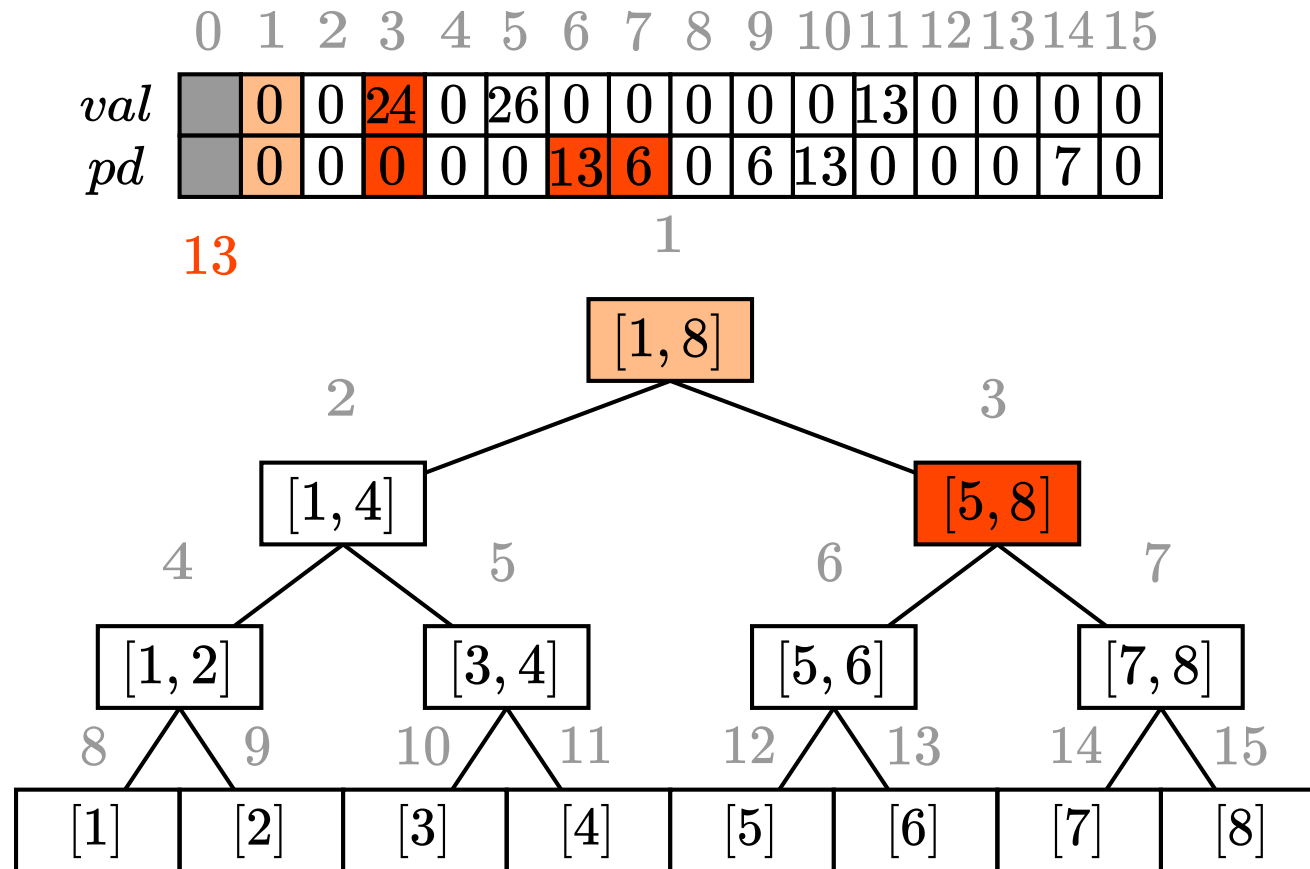
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<i>val</i>		0	0	0	0	26	0	0	0	0	0	13	0	0	0	0
<i>pd</i>		0	0	6	0	0	7	0	0	6	13	0	0	0	7	0



BST – Segment trees (query $\Sigma[4, 6]$)



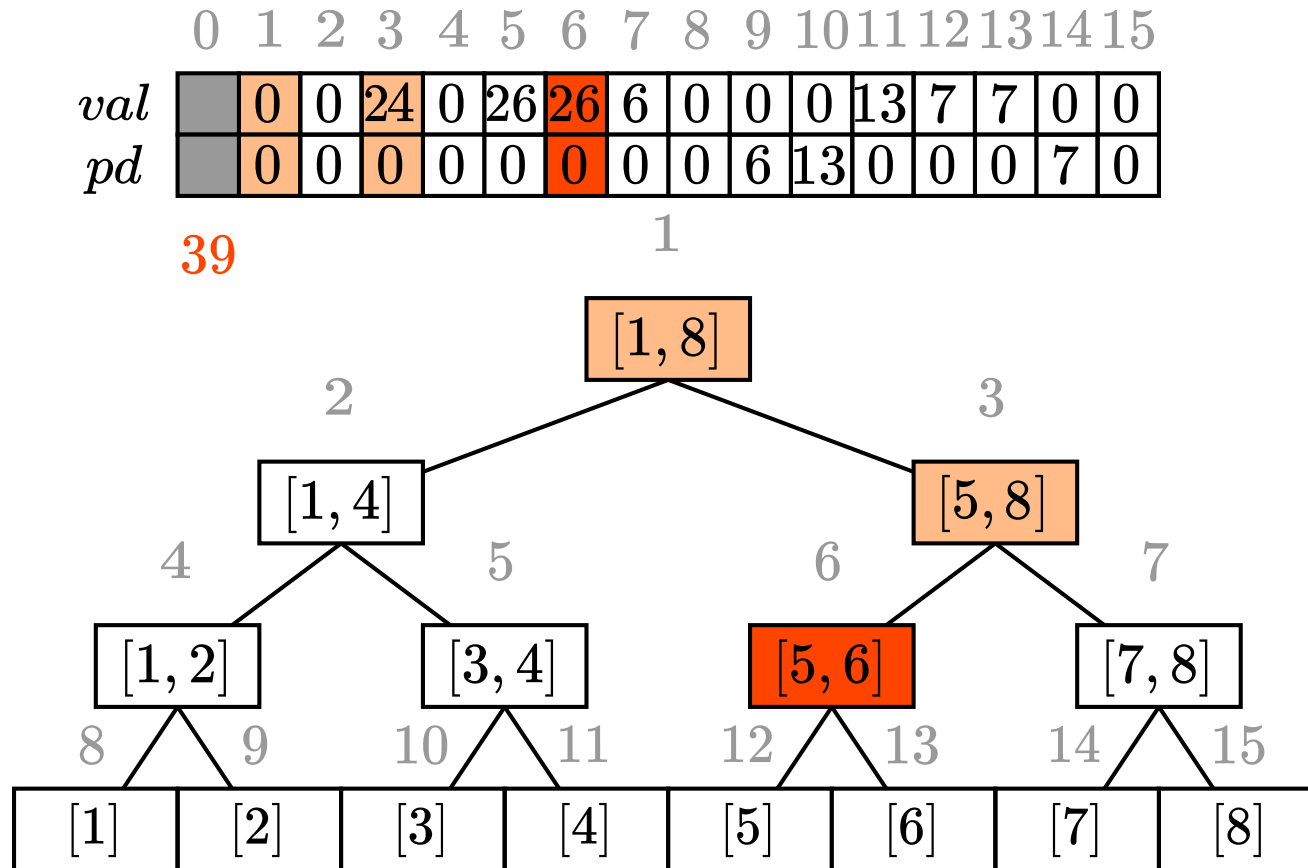
BST – Segment trees (query $\Sigma[4, 6]$)



13 1



BST – Segment trees (query $\Sigma[4, 6]$)



BST – Segment trees analysis



Point queries and updates are straightforwardly logarithmic.

For a range query/update one can observe that we will stop in at most two nodes at each depth.

Pushdown values add a constant overhead to both.

BST – Dynamic segment trees



Sometimes n is too large to materialise the entire tree.

One can create nodes dynamically when they are needed.

The root always exists, when we want to go to a child we first check if it exists and create it if not.

That way our memory usage increases by $\mathcal{O}(\log(n))$ at each update/query.

BST – Multi-dimensional segment trees



Segment trees generalise to hyperrectangles in multi-dimensional spaces.

The idea is that we have the top-level segment tree that holds *other segment trees* in each node.

For example, the node for $[5, 8]$ holds a segment tree for all rectangles whose X -axis dimension is exactly $[5, 8]$.

When trying to update or query a value in a node in the top-level tree we go into its tree and run the update/query on that tree on the second dimension.

This generalises to k dimensions with even more nested trees.

The time becomes $\mathcal{O}(\log^k n)$. Dynamic allocation is basically necessary.

B-trees are a fundamental search structure in database indices.

They don't really apply in competitive programming, but we're the database chair so it'd be weird to not even mention them.

In a B-tree of order m :

- a non-leaf node (called *internal*) with k children has $k - 1$ keys.
- every internal node has between $\left\lfloor \frac{m}{2} \right\rfloor$ and m children;
- all leaves have the same depth;

In short, they're search trees where nodes hold more than 2 children so you need to search through the node's keys to know where to recurse.

The tree is balanced with splits, joins, and rotations.

When inserting into a node would make it contain m keys, it is split into two siblings and the midpoint is inserted as a key to the parent.

When deleting two siblings are joined if they have fewer than $\left\lfloor \frac{m}{2} \right\rfloor$ nodes. Otherwise, one key can be moved from one to the other through the parent.

B-trees are used in databases by setting the order such that a single node's data fits into one filesystem page.

This is because database accesses are¹ I/O bound, and disk I/O always happens in full pages.

Databases often utilise B+-trees, which only store full values in leaves while internal nodes maintain keys for navigation and oftentimes aggregates.

¹Well, *were*, historically. The landscape now is much more complex, and you can learn about it more by e.g. writing your thesis with us 😊. The B-tree remains a good choice but now due to cache misses in main memory.

See you next week

DUN: 29.10.2026, 12:00 PM

Good luck!

