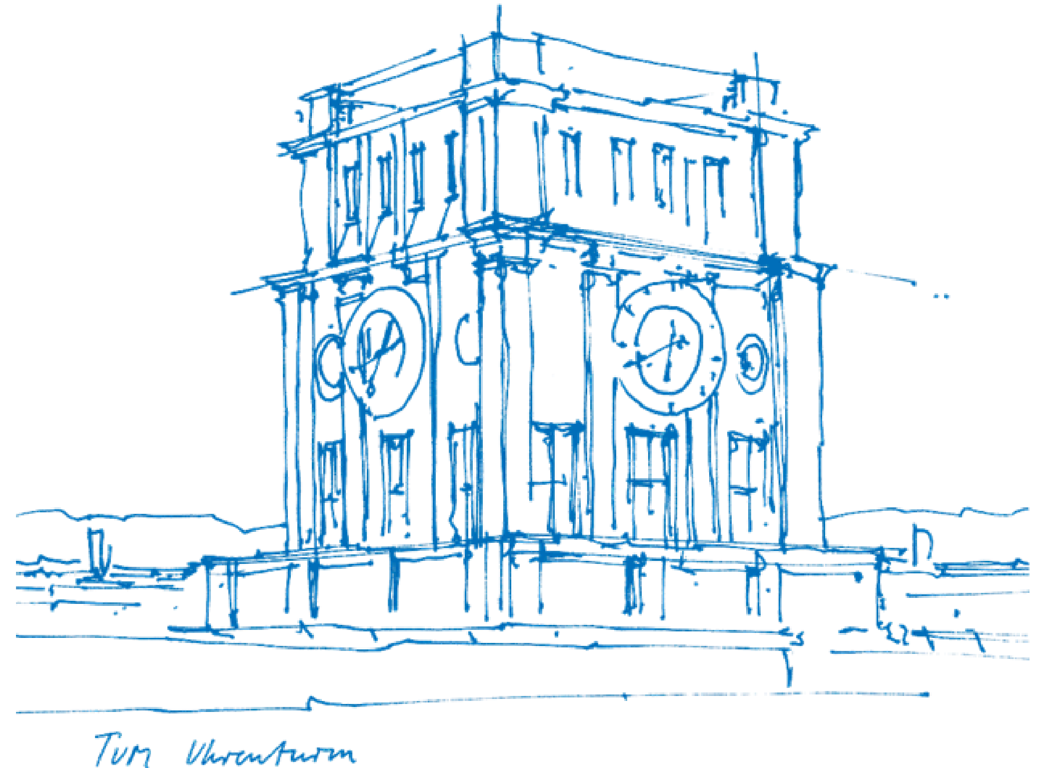
AACPP WiSe 2025/26

Class 2: Binary Search Trees

Mateusz Gienieczko

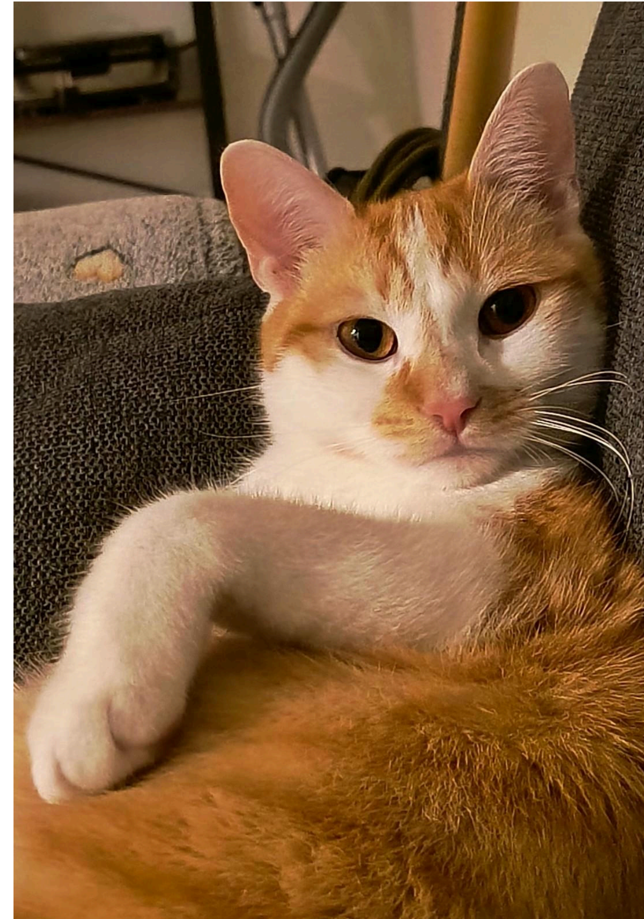
School of Computation, Information and Technology
Technical University of Munich

2025.11.04



Second problem

Second deadline – 19.11.2025, 12:00 AM.



CTML – CaTML

We are given a string s, $|s| = n$, of opening and closing brackets.

We want to split it into k segments such that the sum of their values is minimal.

The value of a segment is the number of substrings that are correct bracketings.

CTML – Bracketings

Correct bracketings have two equivalent definitions.

Recursive – given in the task.

Iterative – consider $<$ as $+1$ and $>$ as -1, then compute the prefix sum.
Bracketing is correct iff the sum never falls below zero and is zero at the end.

CTML – Basic DP

Direct DP – let $\mathrm{DP}_{i,j}$ be the minimal solution for the prefix of s of length j and i segments. $\mathrm{DP}_{0,0} = 0$ and the solution is $\mathrm{DP}_{n,k}$.

$$\mathrm{DP}_{i,j} = \min_{0 \le x < j} \left(\mathrm{DP}_{i-1,x} + c(x+1, j)\right)$$

where $c(a, b)$ is the number of correct bracketings in $s[a..b]$.

$\mathcal{O}(nk)$ states, c can be precomputed in $\mathcal{O}\left(n^2\right)$, total $\mathcal{O}\left(n^2 k\right)$. 2 points.

CTML – Calculating value

First compute for each $s[a..b]$ if it is a correct bracketing in $\mathcal{O}(n^2)$.

Then compute $c(a,b)$ in order of increasing length $(b-a+1)$.

$$c(a,b) = c(a+1,b) + c(a,b-1) - c(a+1,b-1) + \text{correct}_{a,b}$$

CTML – Knuth's optimisation

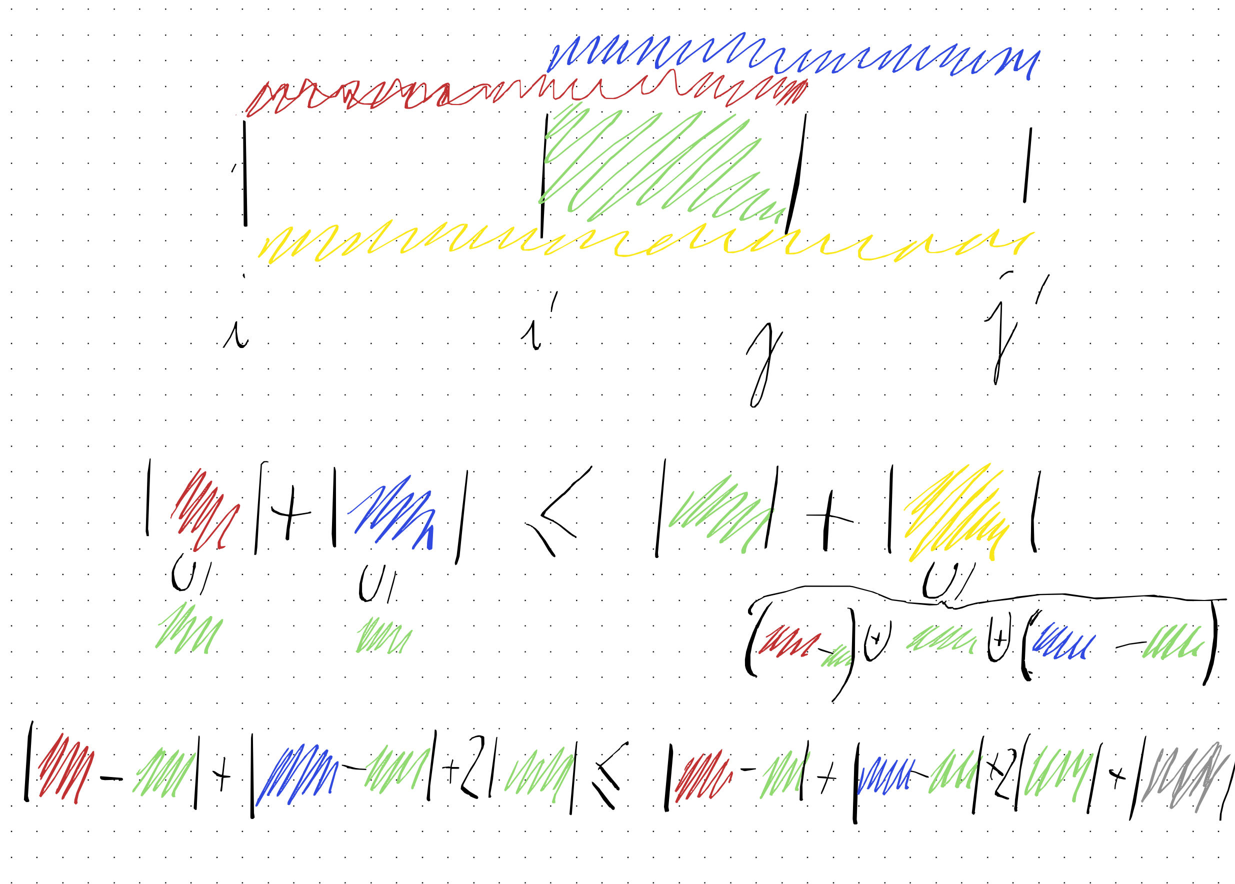
Knuth's optimisation gets us down to $\mathcal{O}(n^2)$ and gives 4 points.

As a reminder, we need to prove that the optimal splitting point $x_{i,j}$ satisfies:

$$x_{i-1,j} \le x_{i,j} \le x_{i,j+1}$$

It's easy to convince ourselves of that with handwaving, but we can also prove the quadrangle inequality on $c(a, b)$ (it is obviously monotone).

CTML – Quadrangle inequality



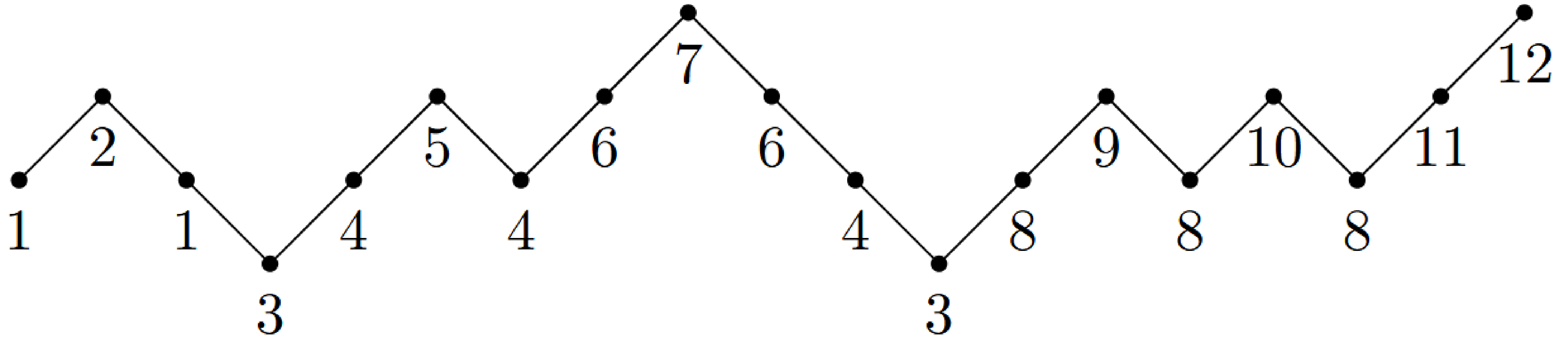
CTML – Faster brackets

To go below n^2 we need to calculate $c(a, b)$ faster.

We will show how to calculate $c(a, b)$ in $\mathcal{O}(b - a + 1)$, but without any recursive relation on c.

Classic approach to bracketing with its geometric interpretation.

CTML – Faster brackets



We group together brackets at the same height that give correct bracketings, i.e. there is no point lower than them in between.

Keep the count of active points of each group. When adding a new point with group i we increase the cost by $\text{cnt}[i]$ and then increase $\text{cnt}[i]$.

CTML – Divide and Conquer

Based on the $x_{i,j} \leq x_{i,j+1}$ property we can also apply divide-and-conquer.

The implementation is again direct. Find $x_{i,\frac{n}{2}}$, recurse. We can compute the groups as above on-the-fly, giving a linear cost for solving each subproblem.

$\mathcal{O}(nk \log n)$ gives the third subtask where $n \leq 100\,000, k \leq 30$.

Sidenote – Combining subtasks

You can combine subtasks to get more points.

Here, for example, with an if condition you can check $k \leq 30$ and then run Divide and Conquer, otherwise run Knuth-optimised. That way you get 6/10 points.

(One person did that, good job :))

CTML – Parametric search

To go faster we need to employ parametric search, since there's $\mathcal{O}(nk)$ states.

Recall – we need the DP function to be convex.

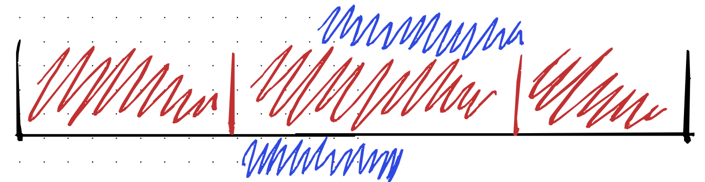
$$\mathrm{DP}_{i,j} - \mathrm{DP}_{i+1,j} \geq \mathrm{DP}_{i+1,j} - \mathrm{DP}_{i+2,j}$$

CTML – Convexity

$$DP_{i,j} - DP_{i+1,j} \geq DP_{i+1,j} - DP_{i+2,j}$$

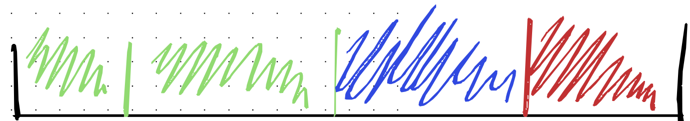
$$2DP_{i+1,j} \leq DP_{i,j} + DP_{i+2,j}$$

$OPT_{3,j}$

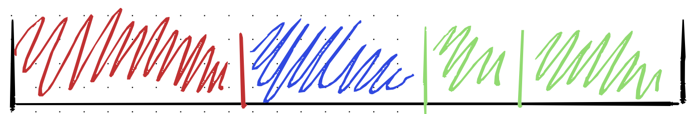


$OPT_{5,j}$

$SOL_{4,j}$



$SOL'_{4,j}$



$$|SOL_{4,j}|, |SOL'_{4,j}| \geq DP_{4,j}$$

$$SOL_{4,j} + SOL'_{4,j} \geq 2DP_{4,j}$$

CTML – Parametric search

Using parametric search, we have a function

$$g(j) = \min_{0 \le x \le k} \mathrm{DP}_{x,j} + x\lambda$$

and we want to minimise it and find for which x it is minimal (solving ties towards lowest x).

We'll use the convex hull trick.

CTML – Convex hull trick

The idea is to keep a *stack* of convex hulls.

Each stack frame corresponds to a currently open block of brackets.

One hull gives us the result for the currently optimal splitting point within the corresponding block.

When we see < we push a new empty hull.

When we see > we take the optimal point from the current hull, pop it, and add the result to the new top of the stack.

CTML – Convex hull trick

Imagine we are looking at one outer group, i.e. we are inside a correct bracketing.

$$< S_1 >< S_2 >< S_3 >< S_4 > \ldots < S_i > \ldots < S_l >$$

Assuming we know how to solve the substrings (recursively), we now are considering whether to insert a separator at a given point at the top level.

The linear functions will be evaluated at x meaning the number of outer bracketings that can contain all the bracketings inside the current block. This naturally multiplies the solution inside the block by the number of outside “contexts” it can be contained within.

CTML – Convex hull trick

$$< S_1 >< S_2 >< S_3 >< S_4 > \dots < S_i > \dots < S_l >$$
$$\hat{}$$

When we open a bracket we are not adding anything to the cost (only closing brackets can create a correct bracketing), but we consider the cost of adding a new separator in this block.

$$g(j)(x) = 0x + \mathrm{sol}(j-1)(i-1) + \lambda$$

Where sol is the minimum of best cost between adding separators before (taken from the current hull) and not adding any, and paying the cost λ for using a separator now.

CTML – Convex hull trick

$$< S_1 >< S_2 >< S_3 >< S_4 > \dots < S_i > \dots < S_l >$$

$$\hat{}$$

When we close a bracket we need the optimal solution for $< S_i >$ which we query from the hull assuming we insert a separator there. Say it is $\text{opt}_{S_i}(i)$.

We keep the tally of blocks before us (i). To the hull we add a new candidate function:

$$f(x) = -ix + \text{opt}_{S_i}(i) + i(i-1)$$

CTML – Convex hull trick

$$f(x) = -ix + \mathrm{opt}_{S_i}(i) + i(i-1)$$

To see why this works, consider we query this function at some later point $j = i + c$. We get:

$$\begin{aligned} -i(i+c) + \mathrm{opt}_{S_i}(i) + i^2 - i &= -i^2 - ic + \mathrm{opt}_{S_i}(i) + i^2 - i \\ &= \mathrm{opt}_{S_i}(i) - (c+1)i \end{aligned}$$

By choosing this point we pay the optimal cost calculated, but we also break the i possible starts for the following c closing brackets (plus our own).

CTML – Summary

The outer parametric search is $\mathcal{O}(\log n)$ and CHT can be done in $\mathcal{O}(n)$, giving $\mathcal{O}(n \log n)$ in total.

One can also do it much more easily with $\mathcal{O}(\log n)$ queries on the hull, giving a total $\mathcal{O}\left(n \log^2 n\right)$, which was also sufficient.

This solution takes fewer than 100 lines of code!

Great example of a task that has a very easy implementation but insanely complicated rationale behind it.

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.

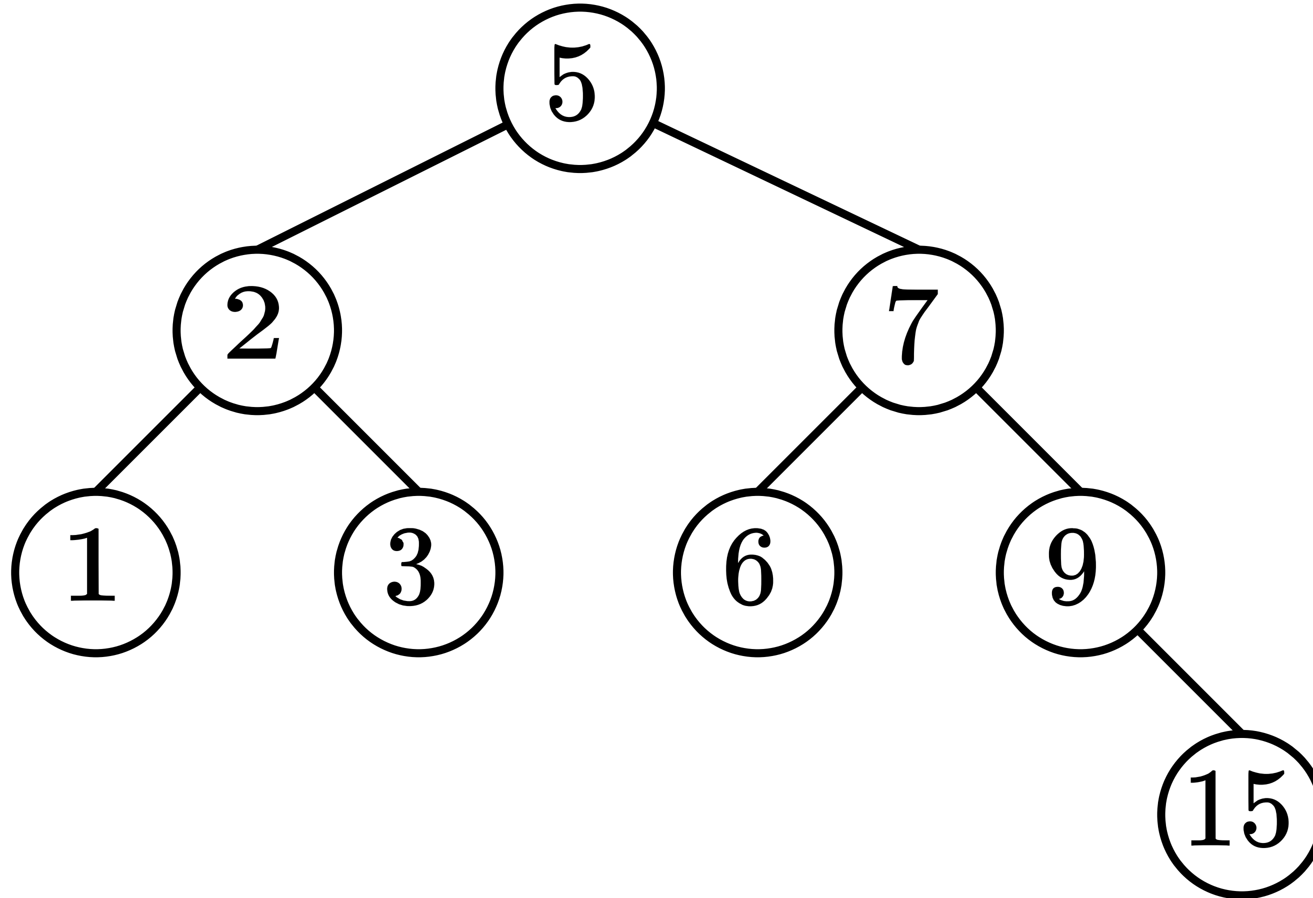
BST – Splay

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

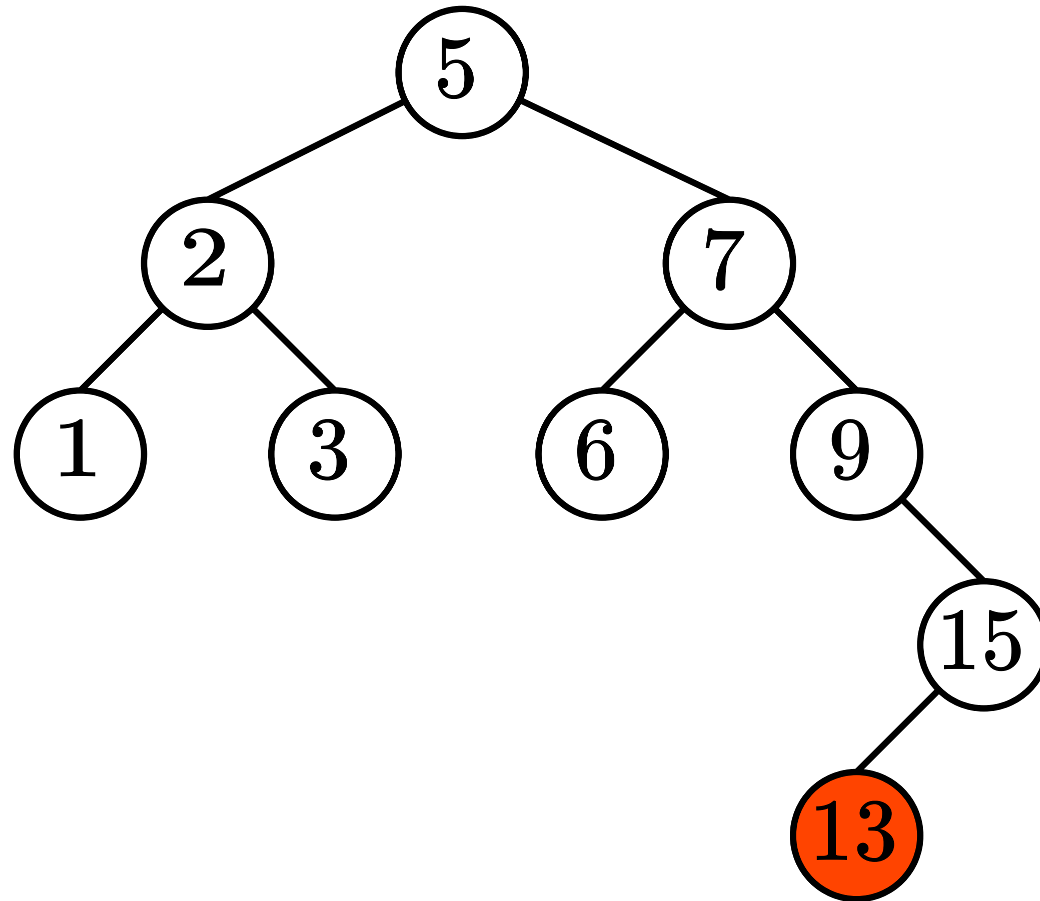
Its runtime is *amortised* $\mathcal{O}(\log n)$ for access and update.

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

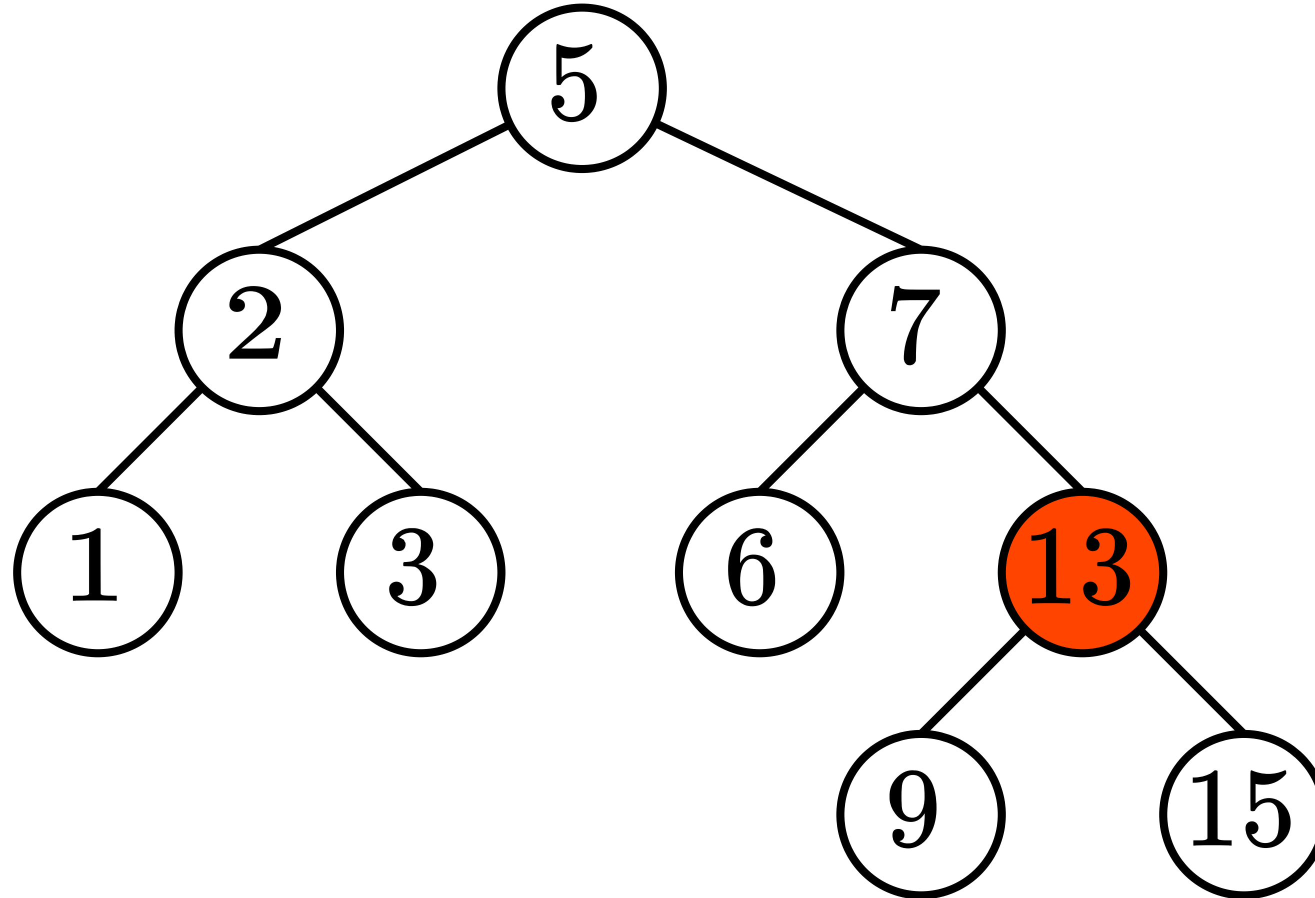
BST – Splay (insertion)



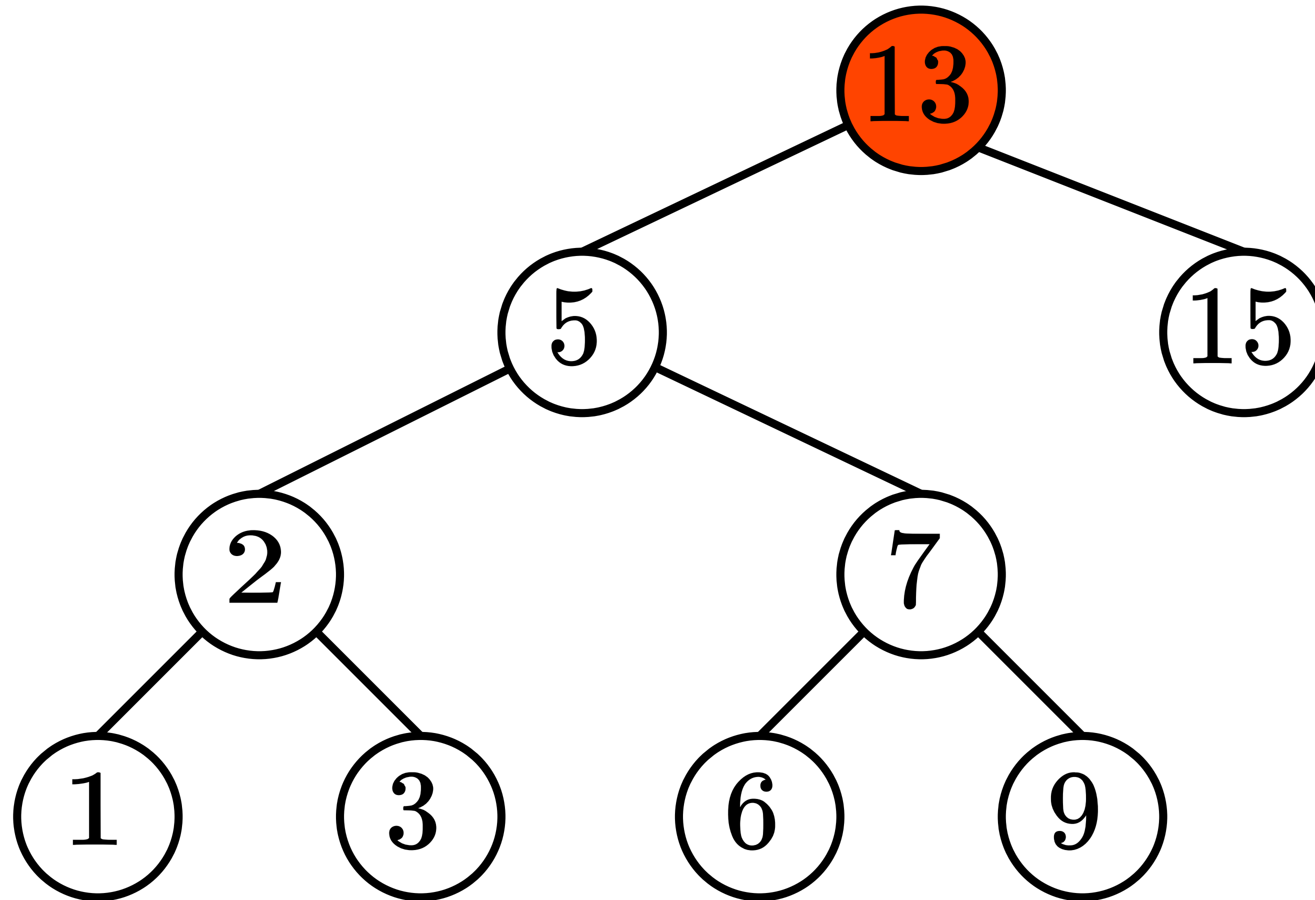
BST – Splay (insertion)



BST – Splay (zag-zig)



BST – Splay (zag-zag)



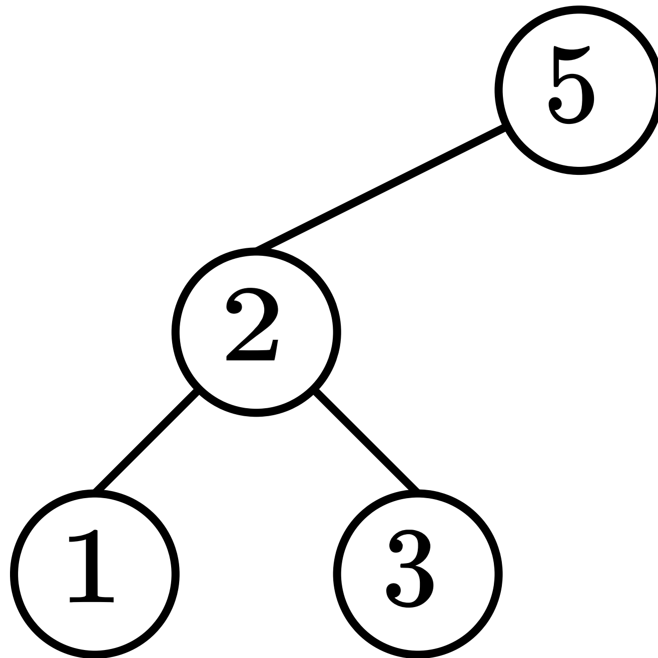
BST – Splay split/join

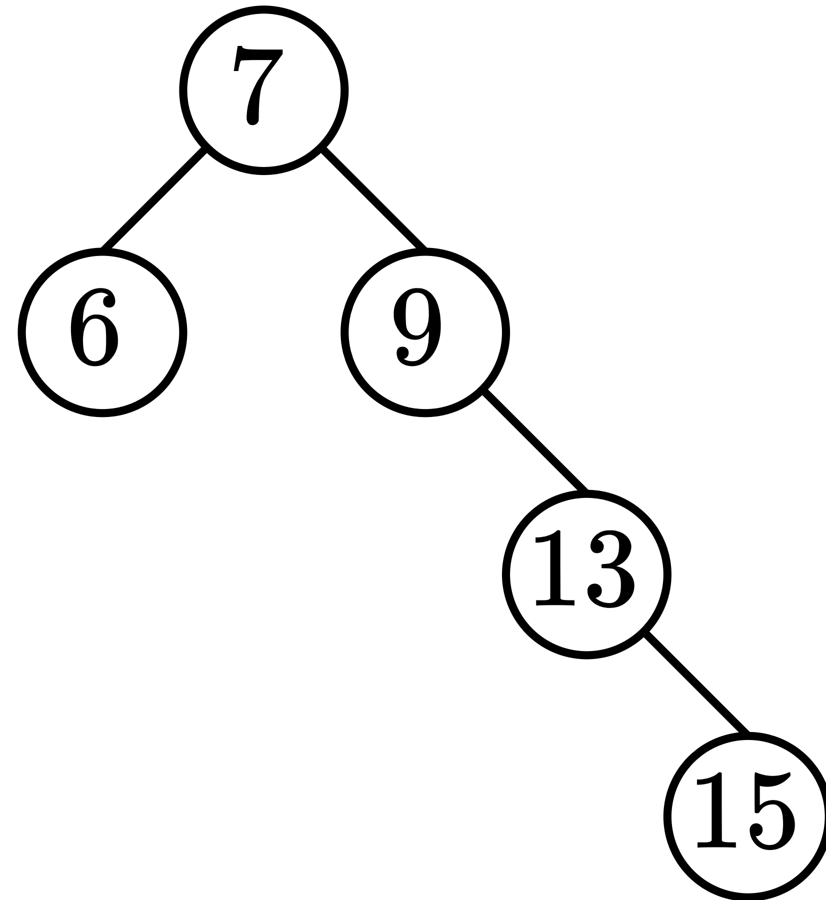
The cool part of splay trees is that we can easily join two splays or split a splay on a given key.

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

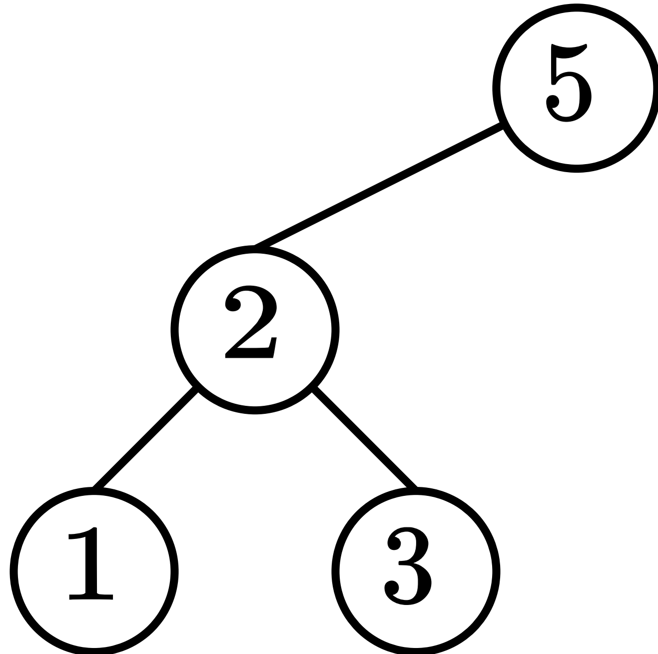
- Insert – split on the key, create a new node as the root.
- Delete – splay the node to the top, remove it, join its children.

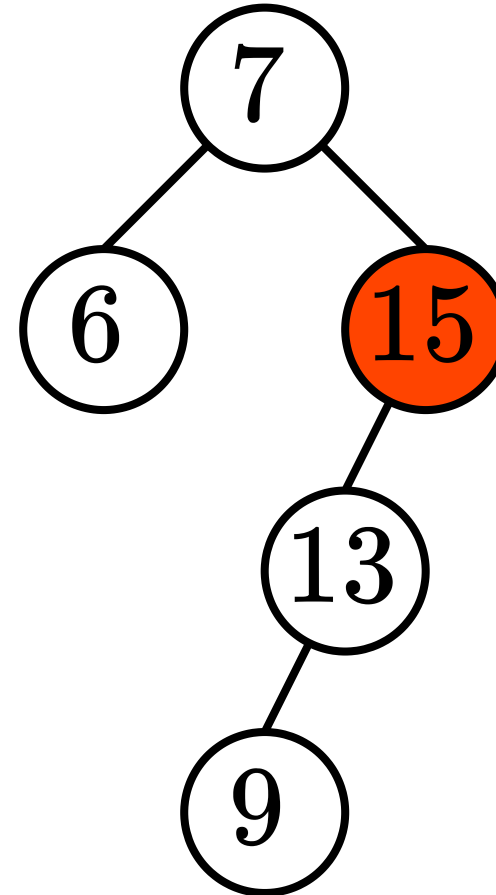
BST – Splay (join)



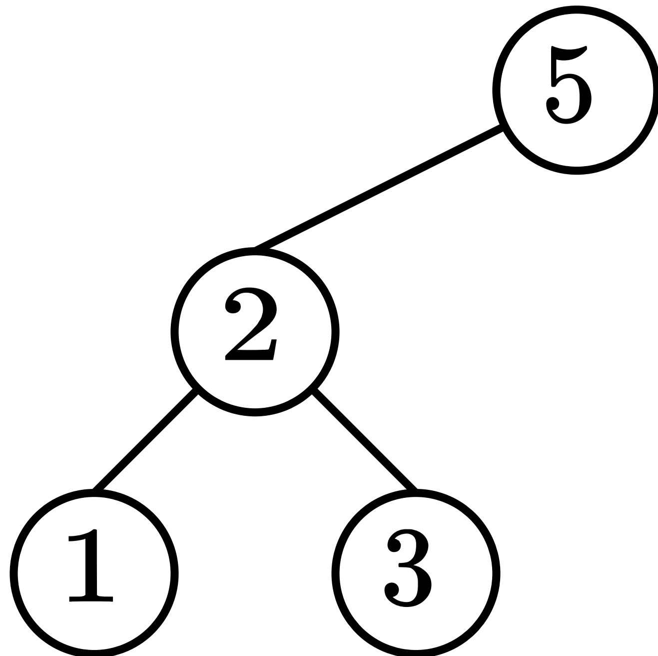


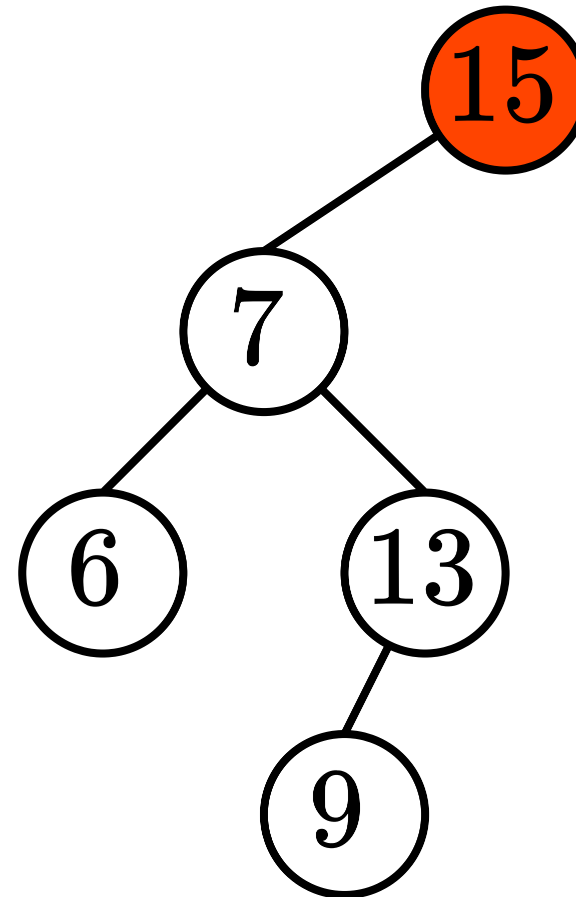
BST – Splay (join)



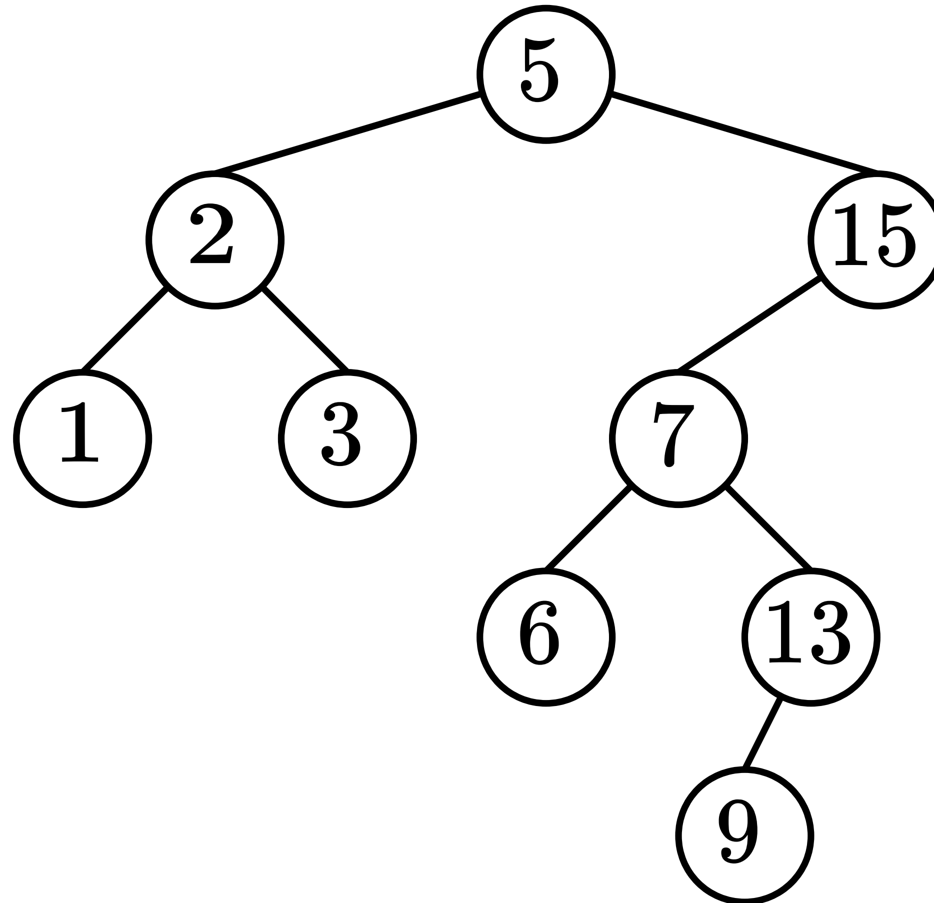


BST – Splay (join)





BST – Splay (join)



BST – Treap

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

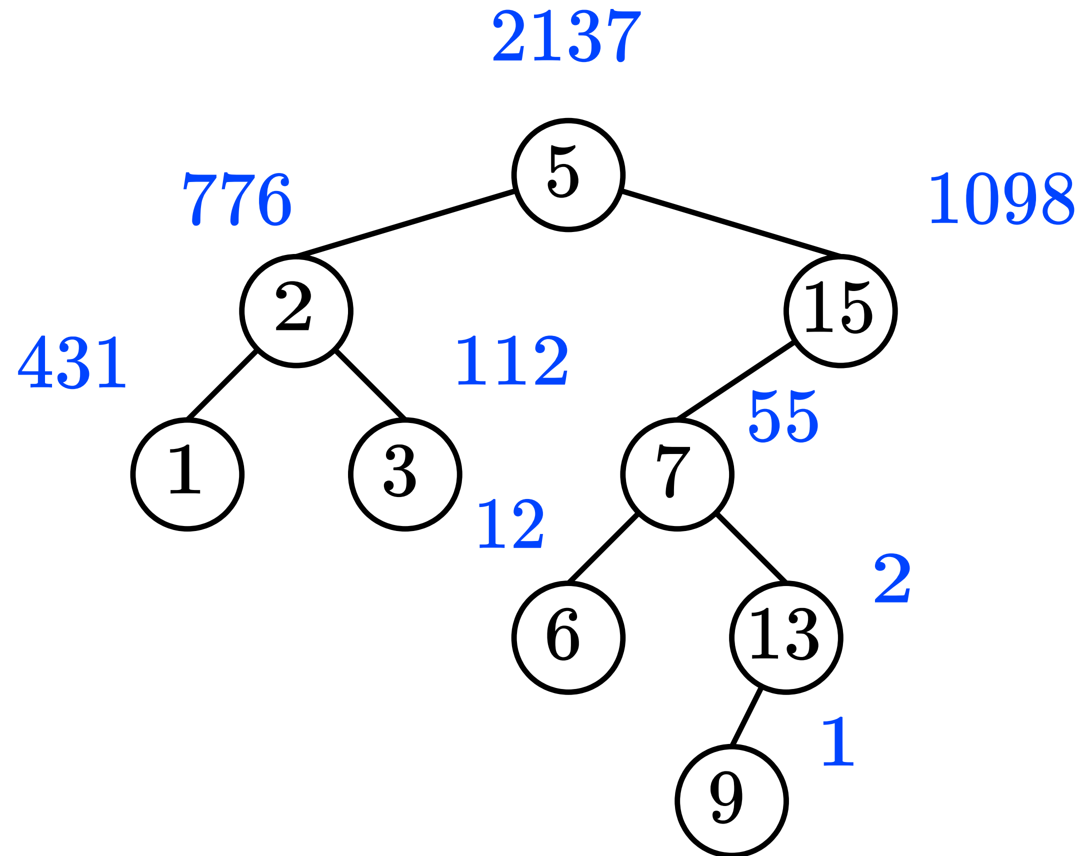
It's based on the following fact:

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

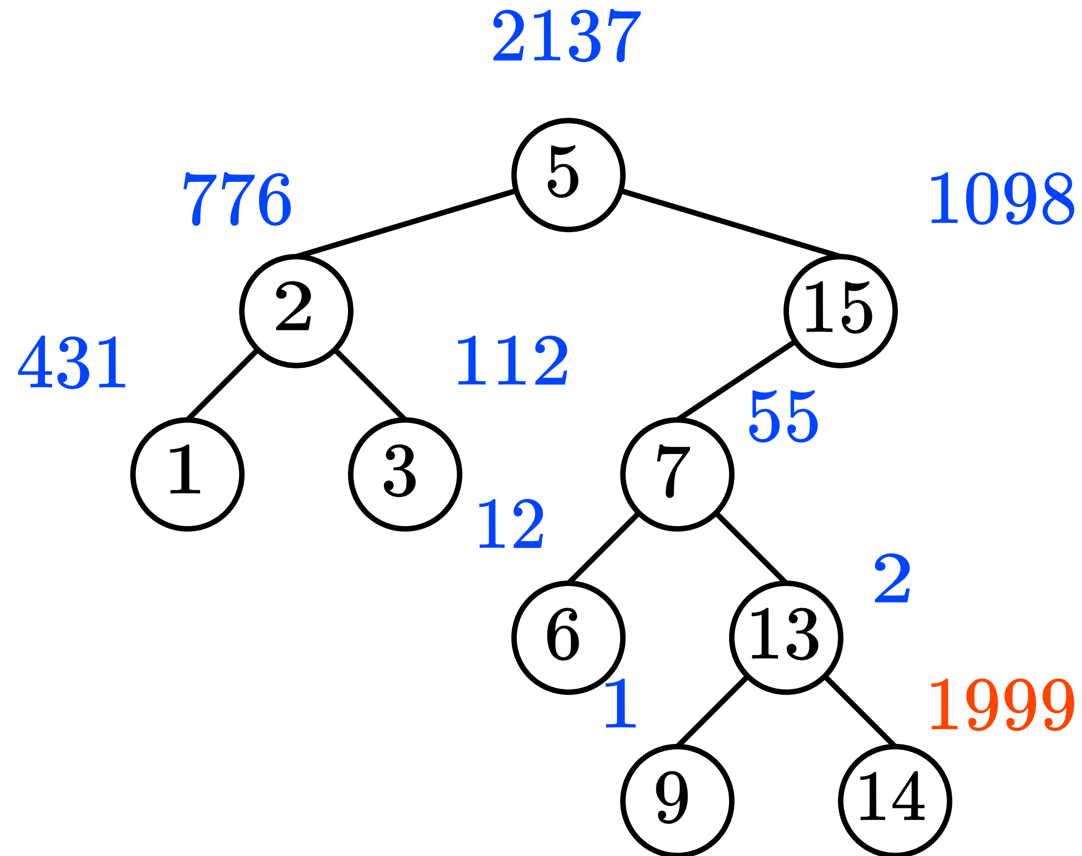
We maintain a BST, but also assign *pseudorandom* weights to each node and maintain the heap invariant (children's weight does not exceed the parent's).

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

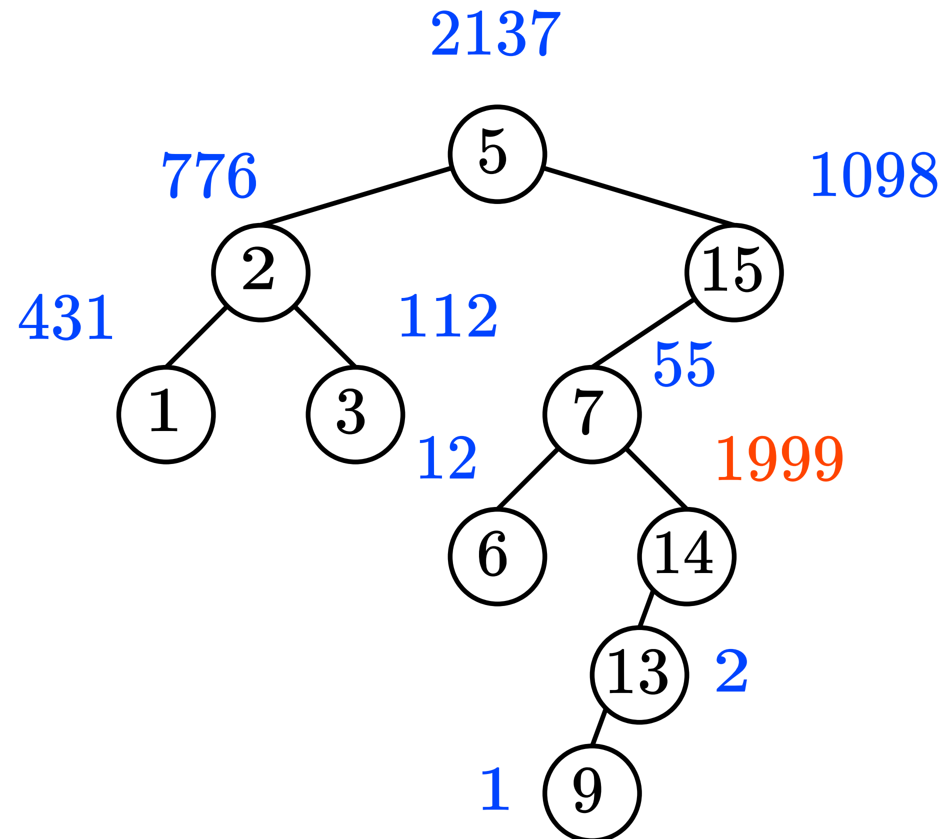
BST – Treap



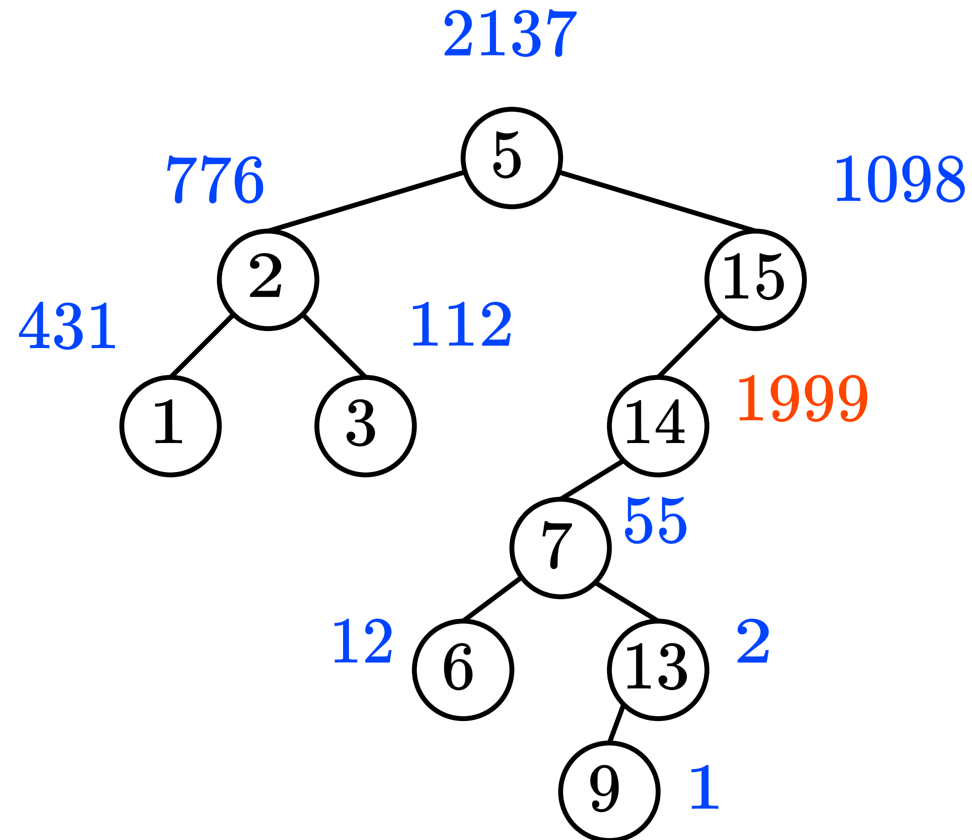
BST – Treap



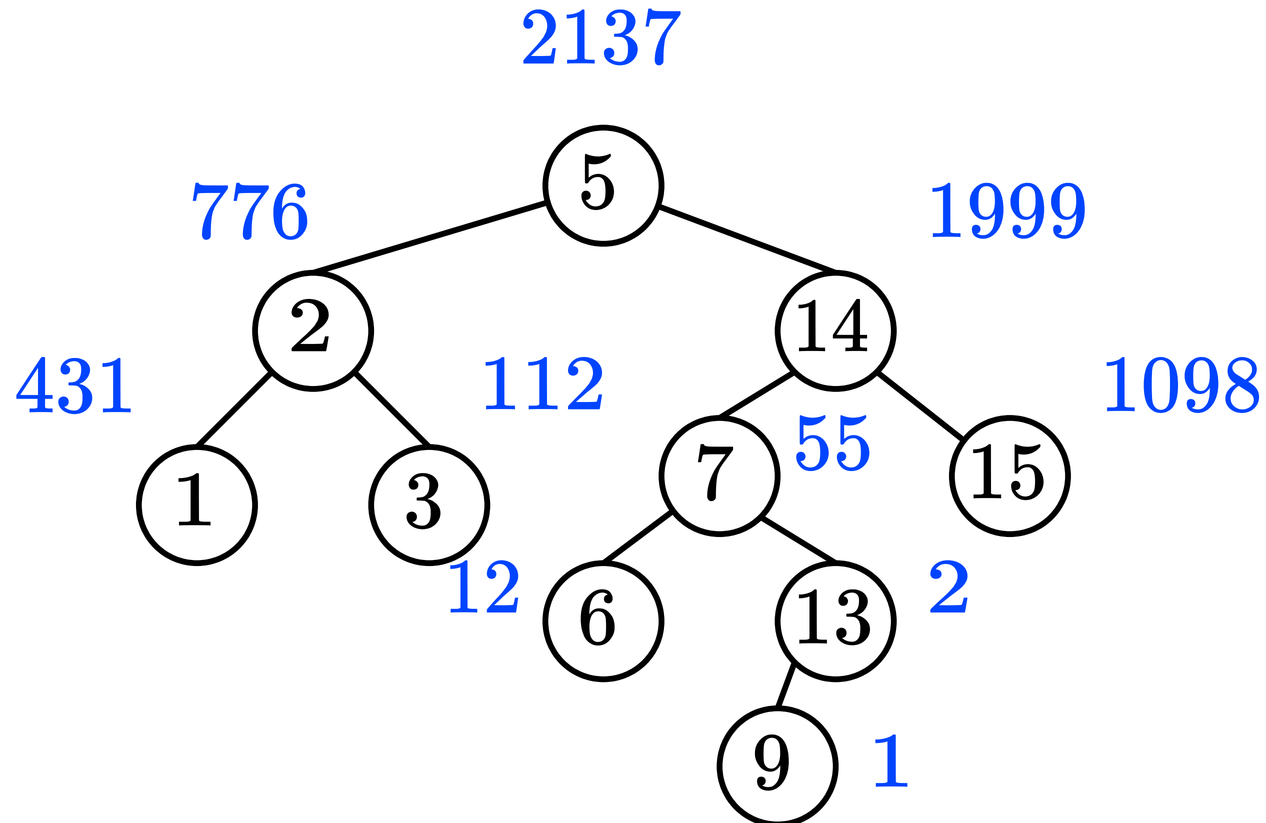
BST – Treap



BST – Treap



BST – Treap



BST – Augmenting BSTs

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:

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

BST – Augmenting BSTs

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.

BST – Implementation notes

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:

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

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

BST – Segment trees

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 $[x, x]$.

BST – Segment trees

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.

BST – Segment trees

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 once 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
```

BST – Pushdown values

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)

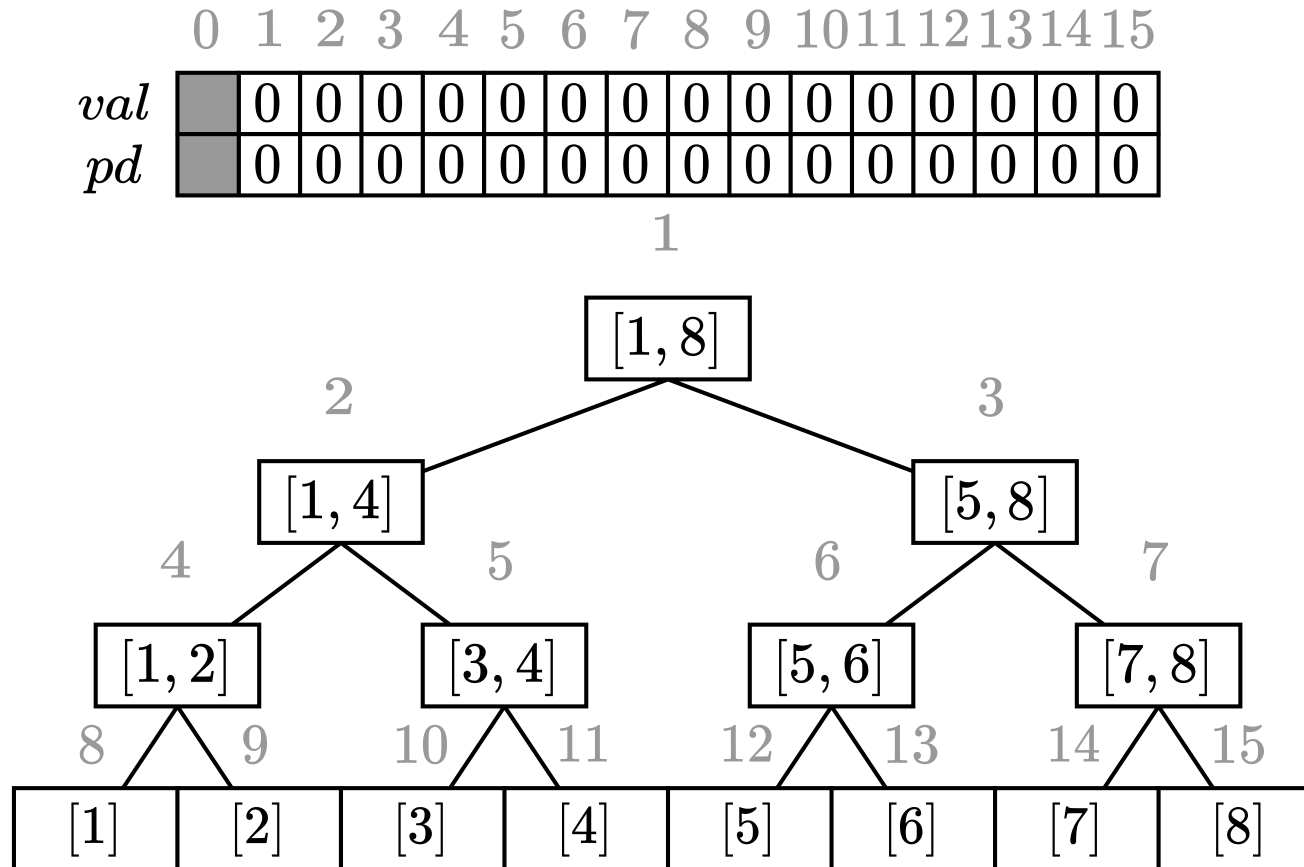
```
fn update_at(q_from, q_to, node, n_from, n_to, value)
  if q_from == n_from and q_to == n_to
    pushdown[node] ⊕= value
  mid = midpoint(n_from, n_to)
  if n_from < mid
    update_at(q_from, min(q_to, mid), 2 * node, n_from, mid, value)
  else
    update_at(max(q_to, mid), q_to, 2 * node + 1, mid + 1, n_to, value)
```

BST – Pushdown values (range query)

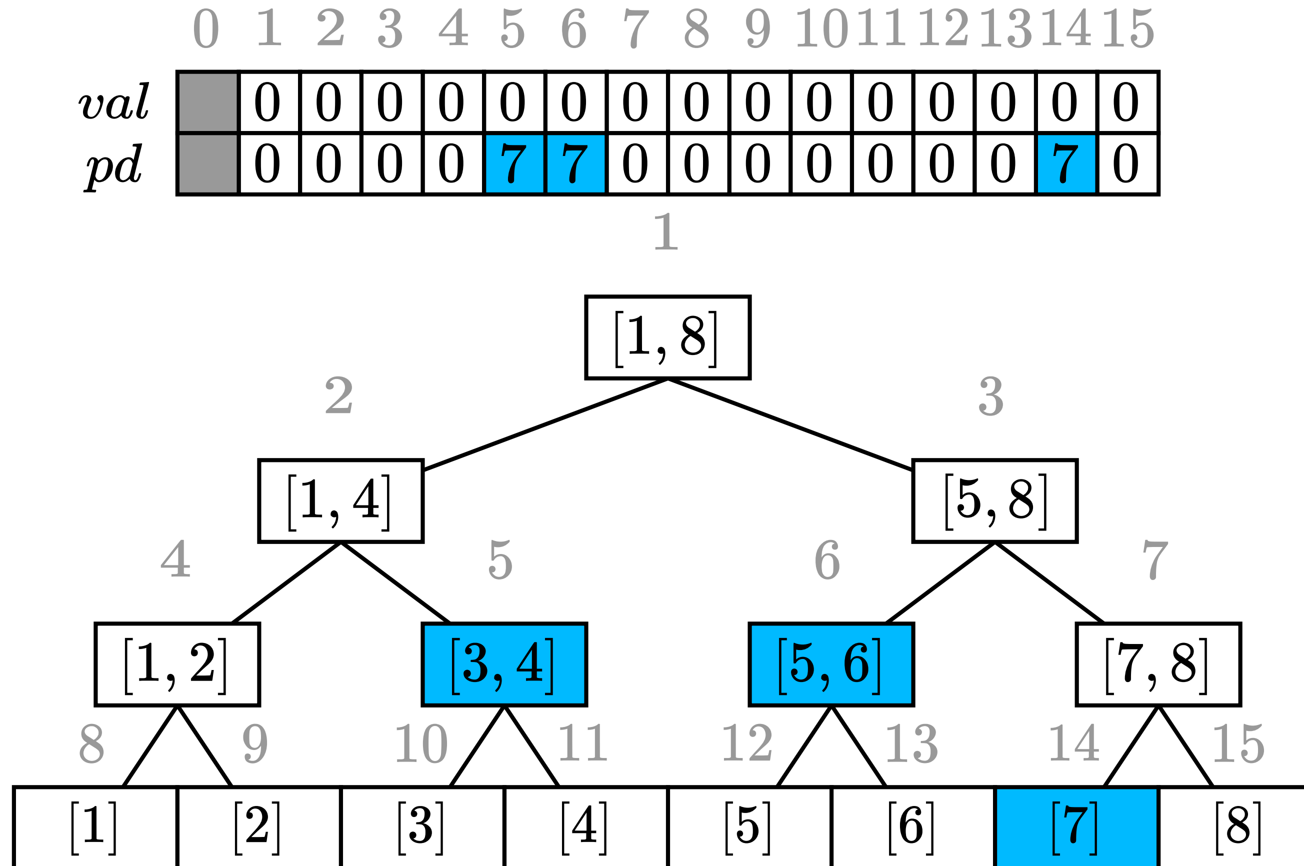
```
fn push_down(node, n_from, n_to)
  tree[node] ⊕= pushdown[node] * (n_to - n_from + 1)
  if n_from != n_to
    pushdown[2 * node] ⊕= pushdown[node]
    pushdown[2 * node + 1] ⊕= pushdown[node]
  pushdown[node] = 0

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
```

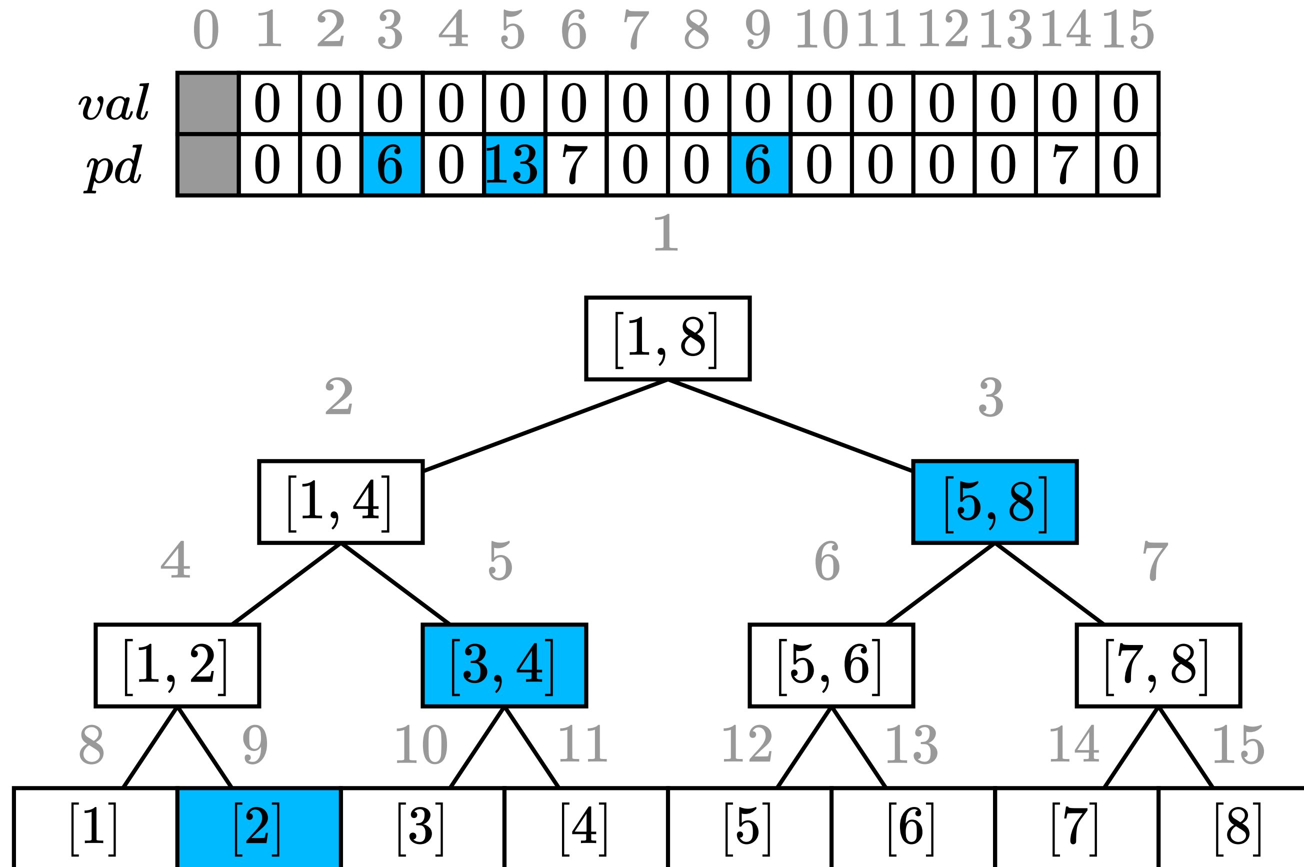
BST – Segment trees



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

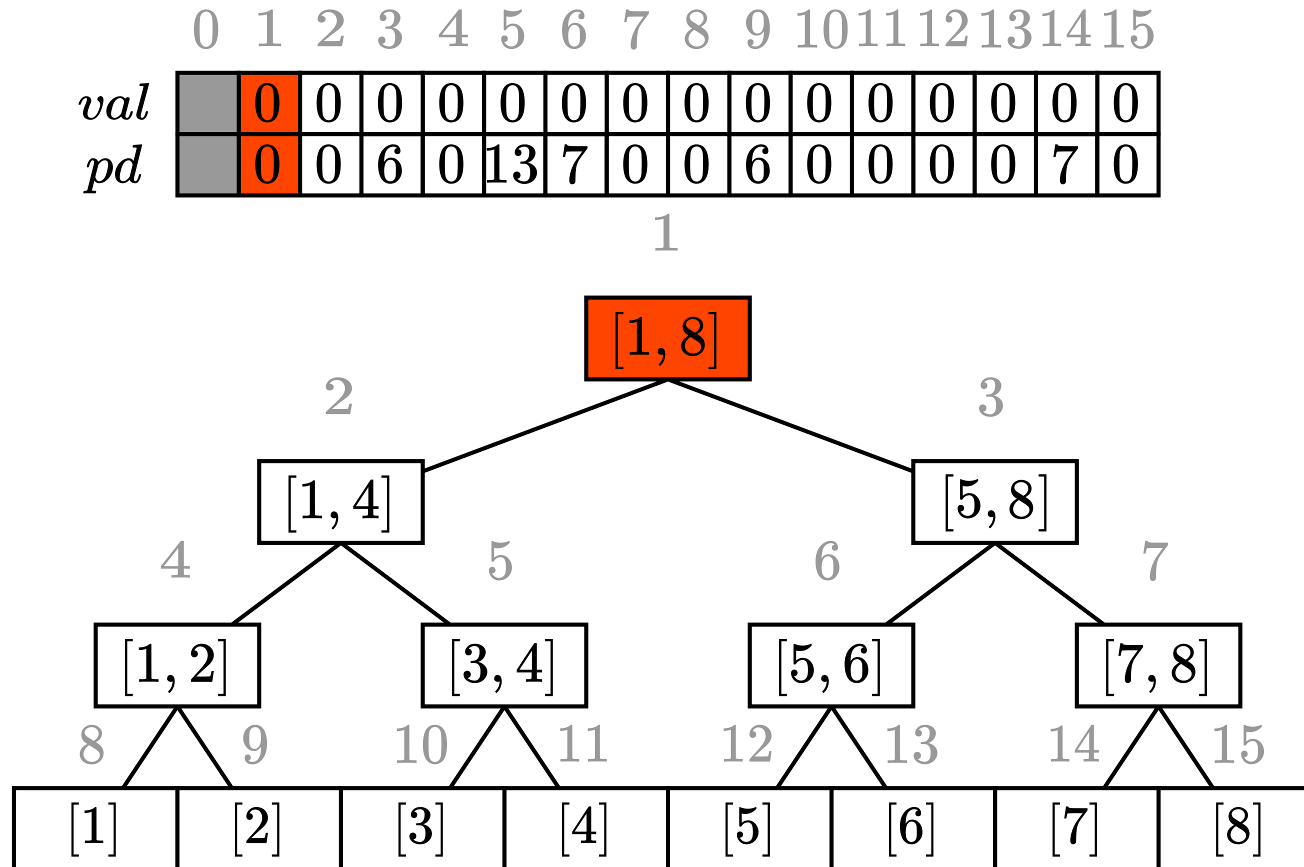


BST – Segment trees (update $[2, 16] + 6$)

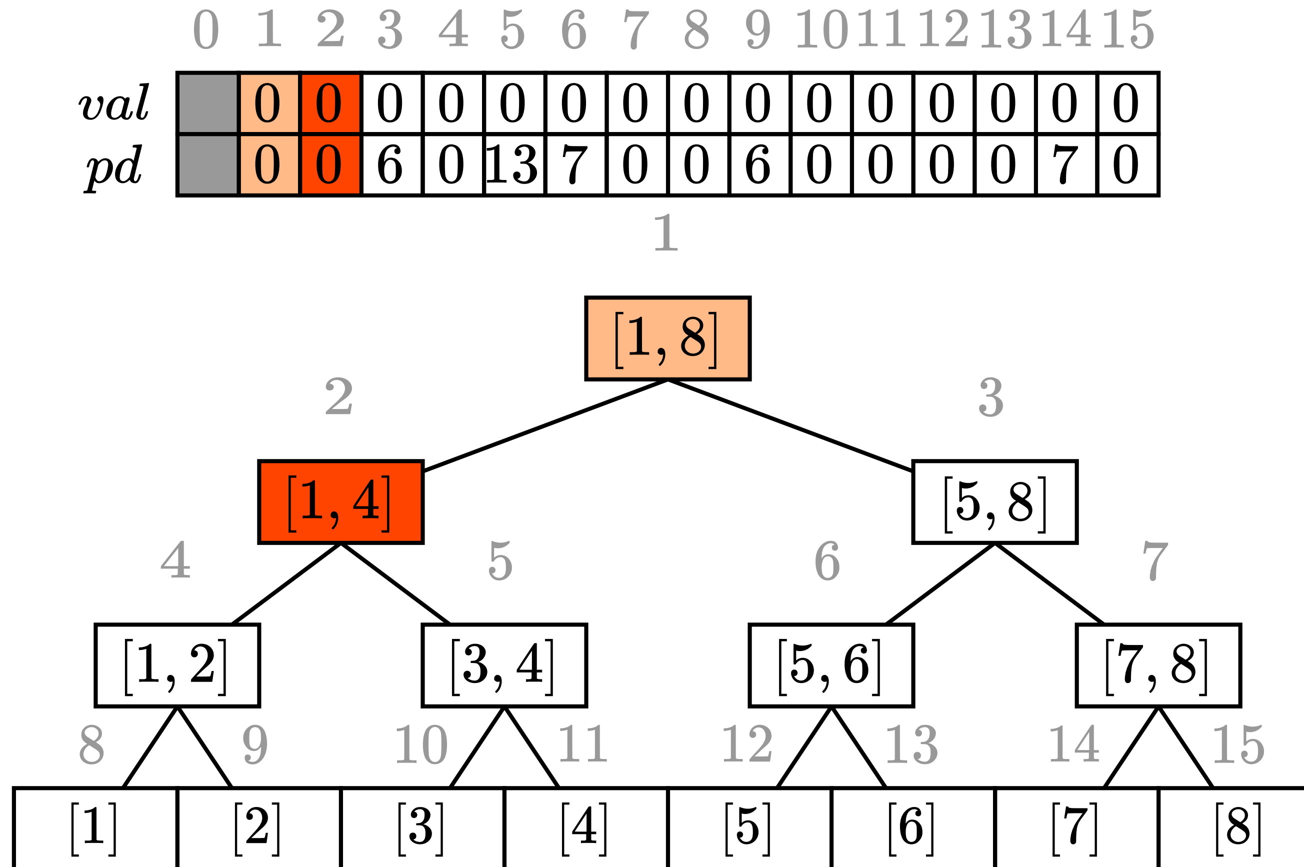


Mateusz Gienieczko

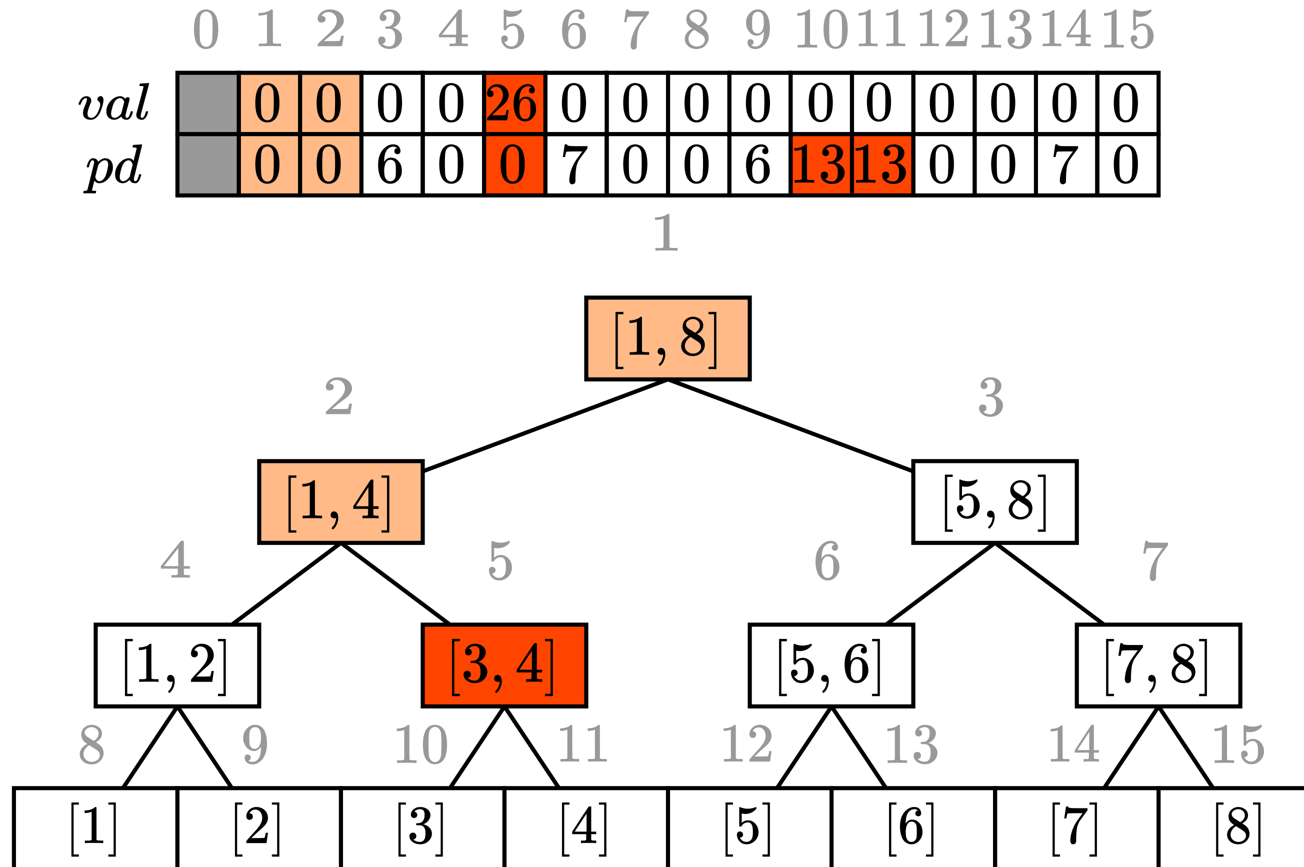
BST – Segment trees (query $\sum[4, 6]$)



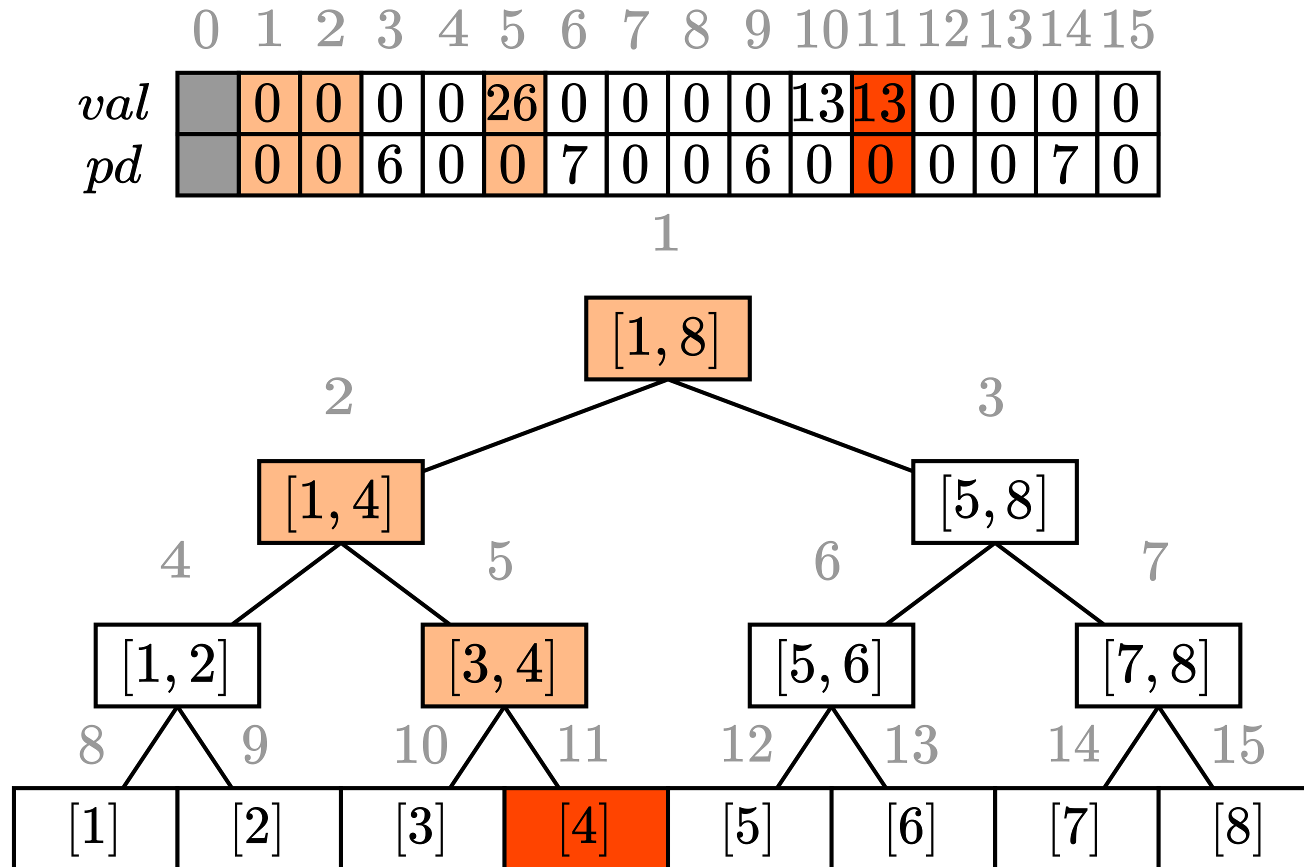
BST – Segment trees (query $\sum[4,6]$)



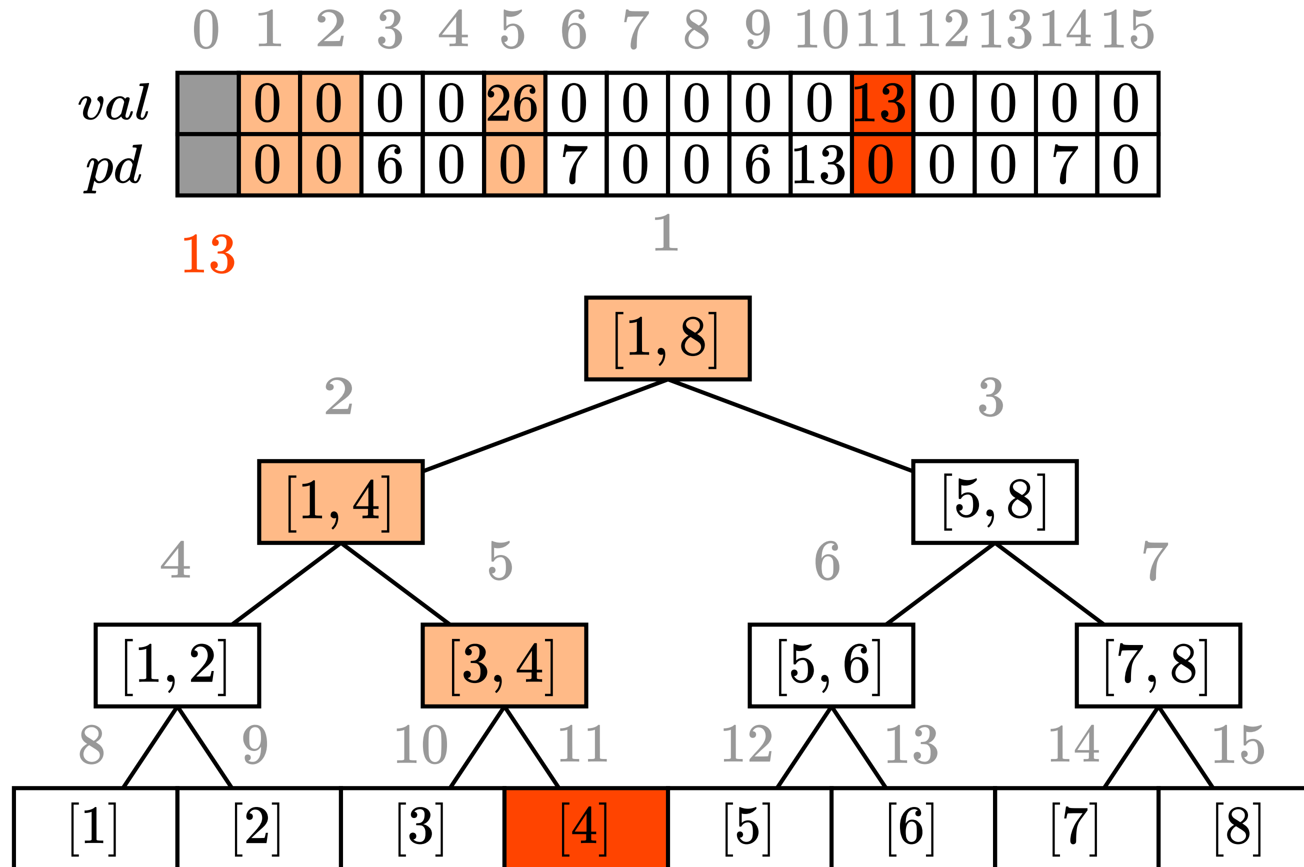
BST – Segment trees (query $\sum[4,6]$)



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

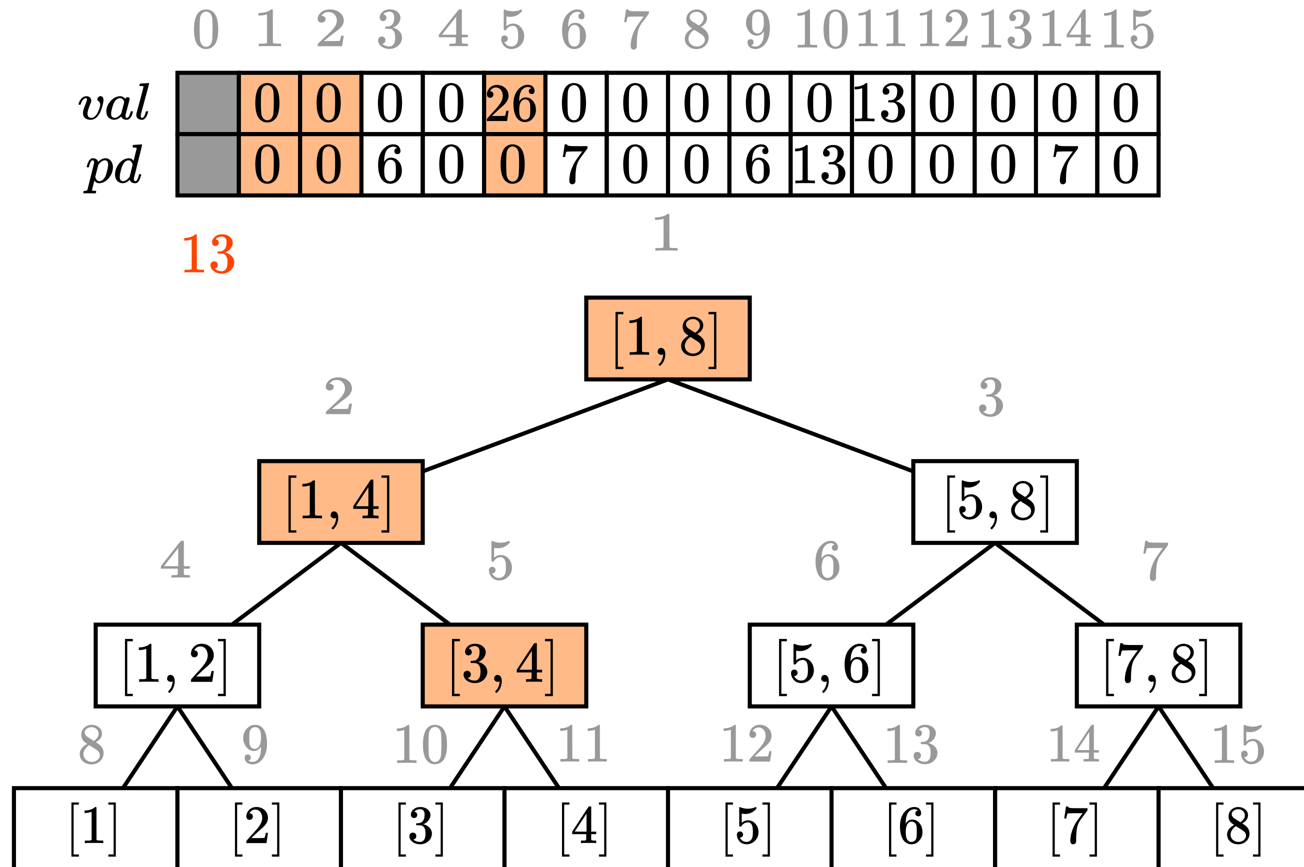


BST – Segment trees (query $\sum[4,6]$)

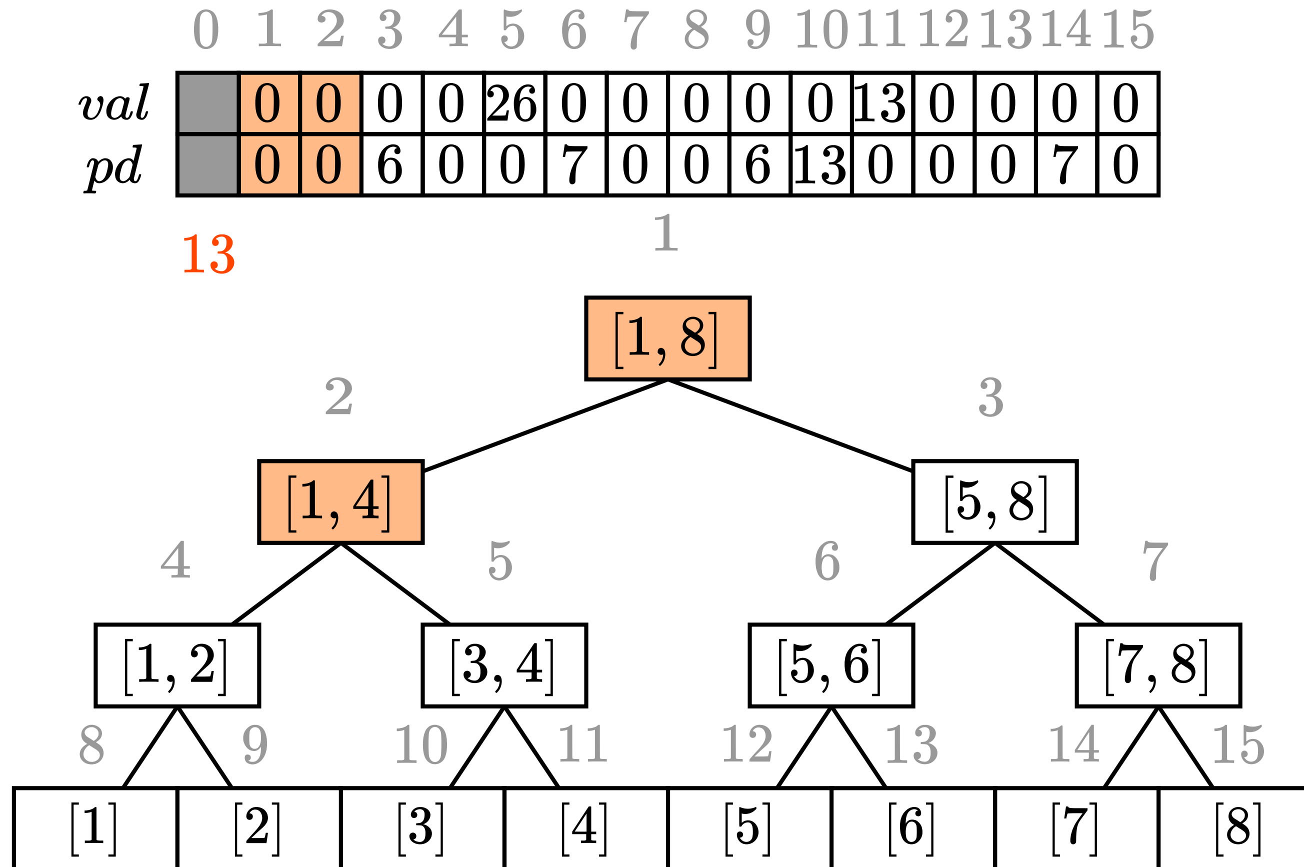


Mateusz Gienieczko

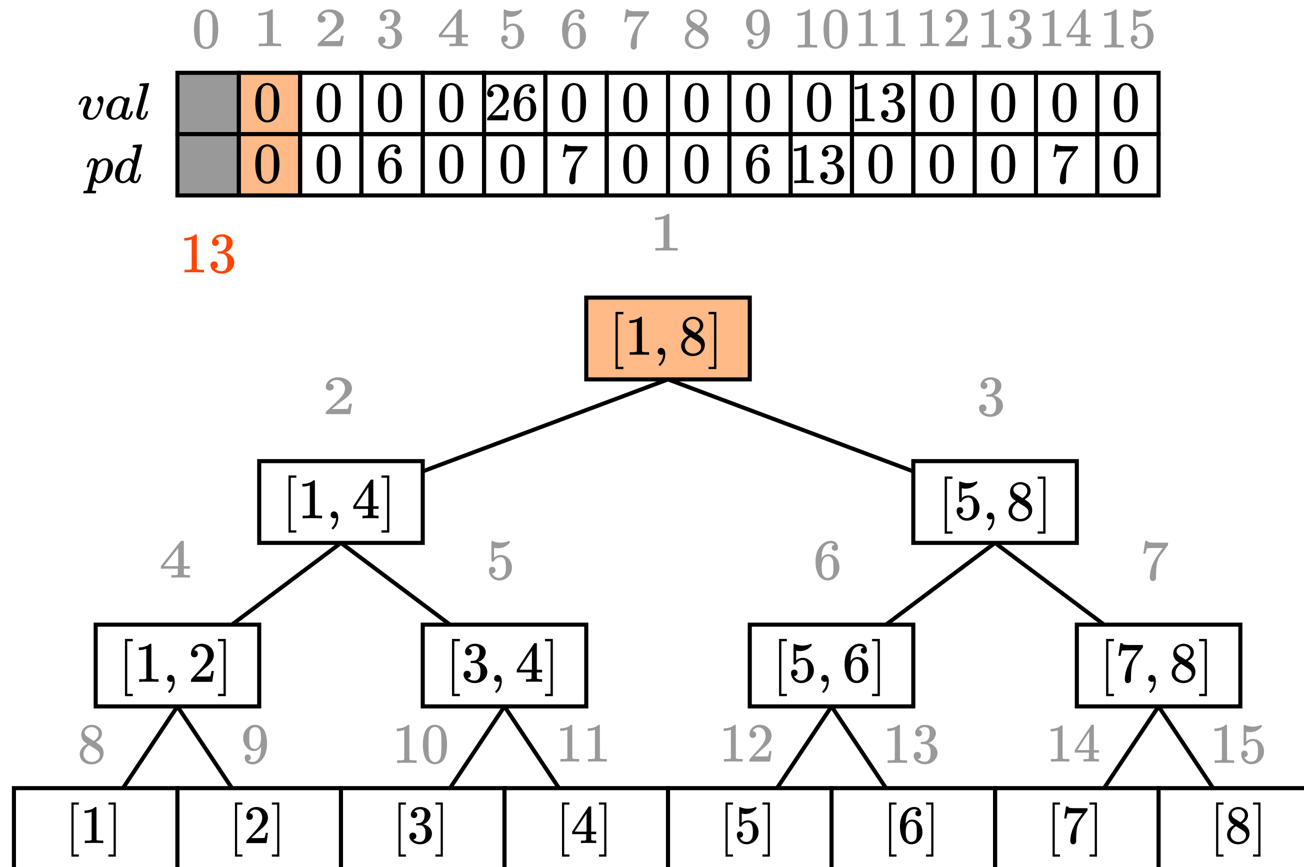
BST – Segment trees (query $\sum[4,6]$)



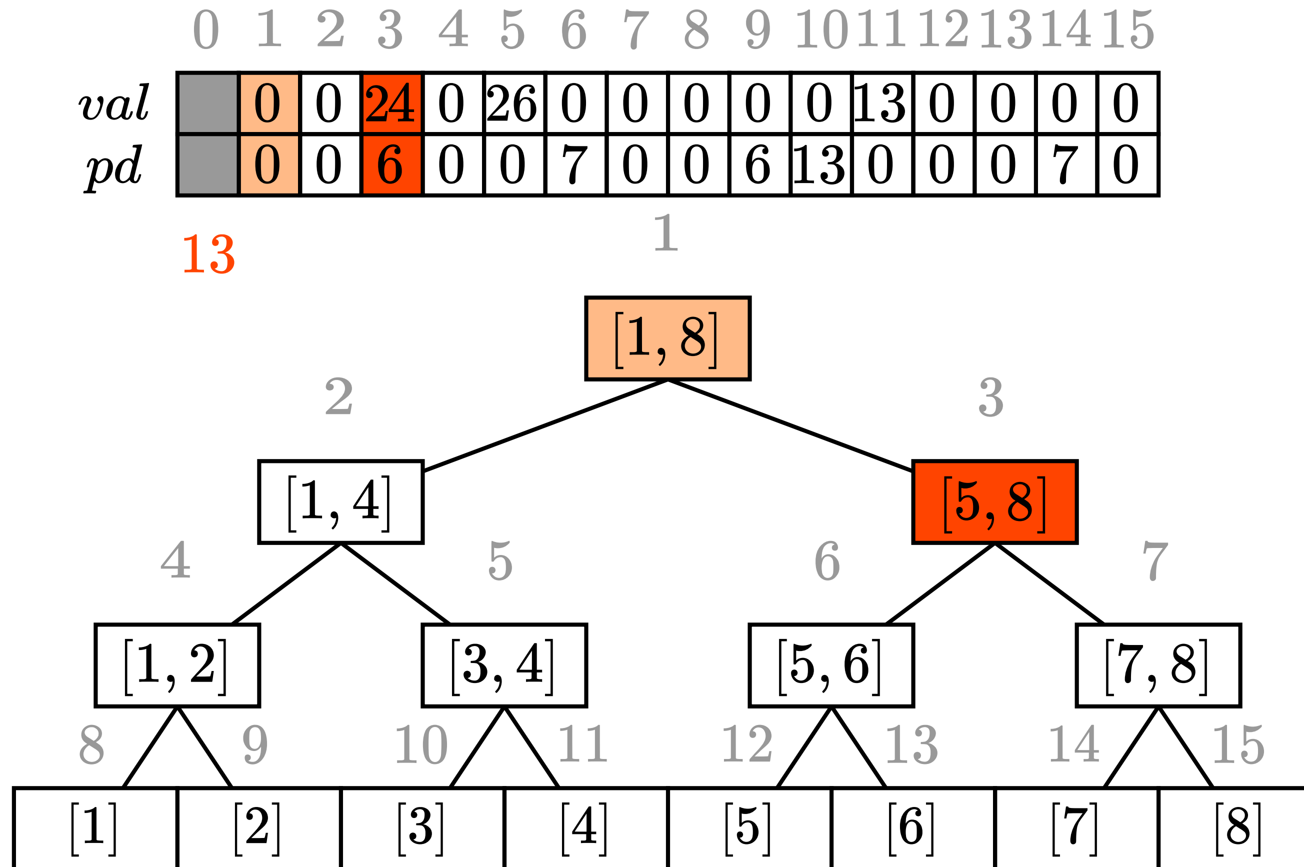
BST – Segment trees (query $\sum[4, 6]$)



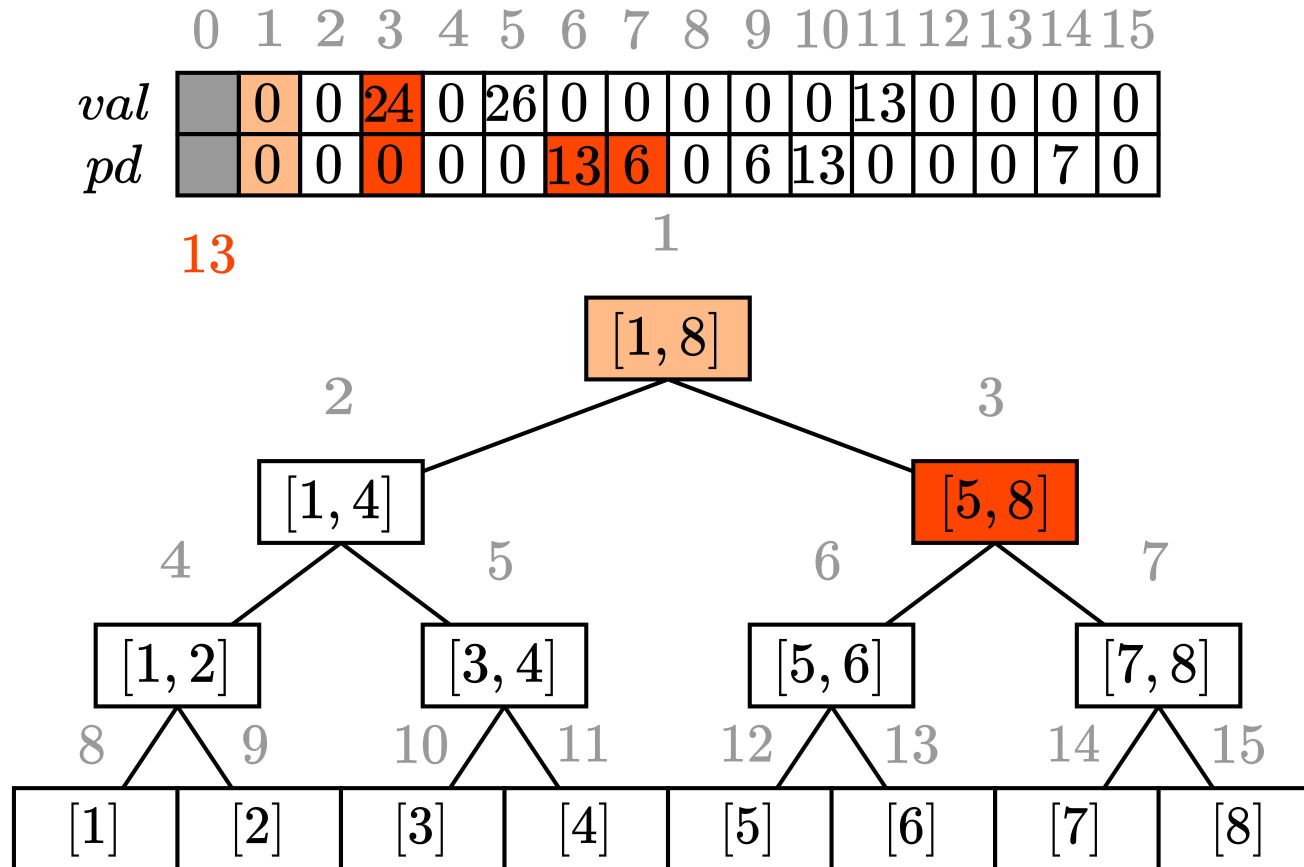
BST – Segment trees (query $\sum[4,6]$)



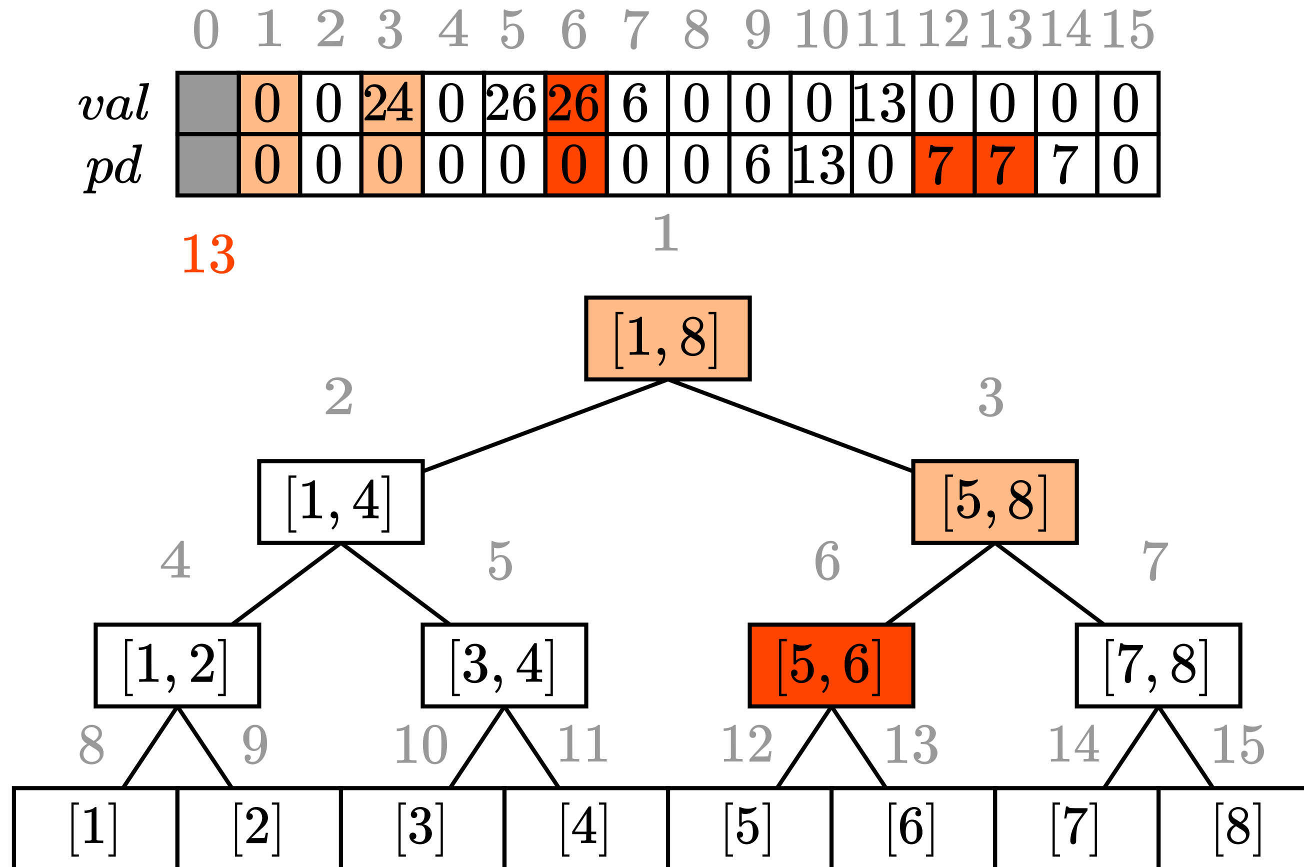
BST – Segment trees (query $\sum[4, 6]$)



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

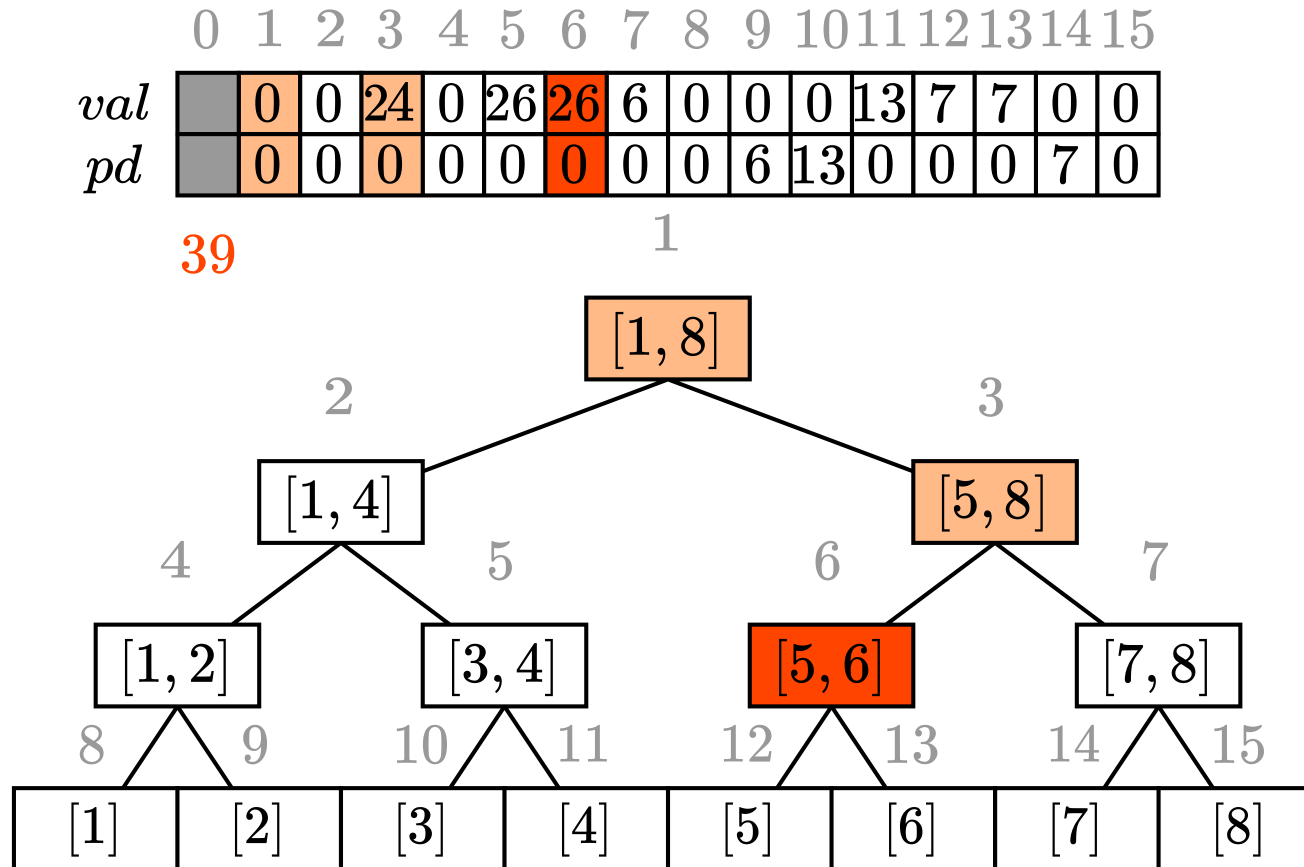


BST – Segment trees (query $\sum[4,6]$)



Mateusz Gienieczko

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



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

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.

B-trees

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

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 are[5] 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.

---

[5] Well, *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

CCC: 19.11.2025, 12:00 AM

Good luck!