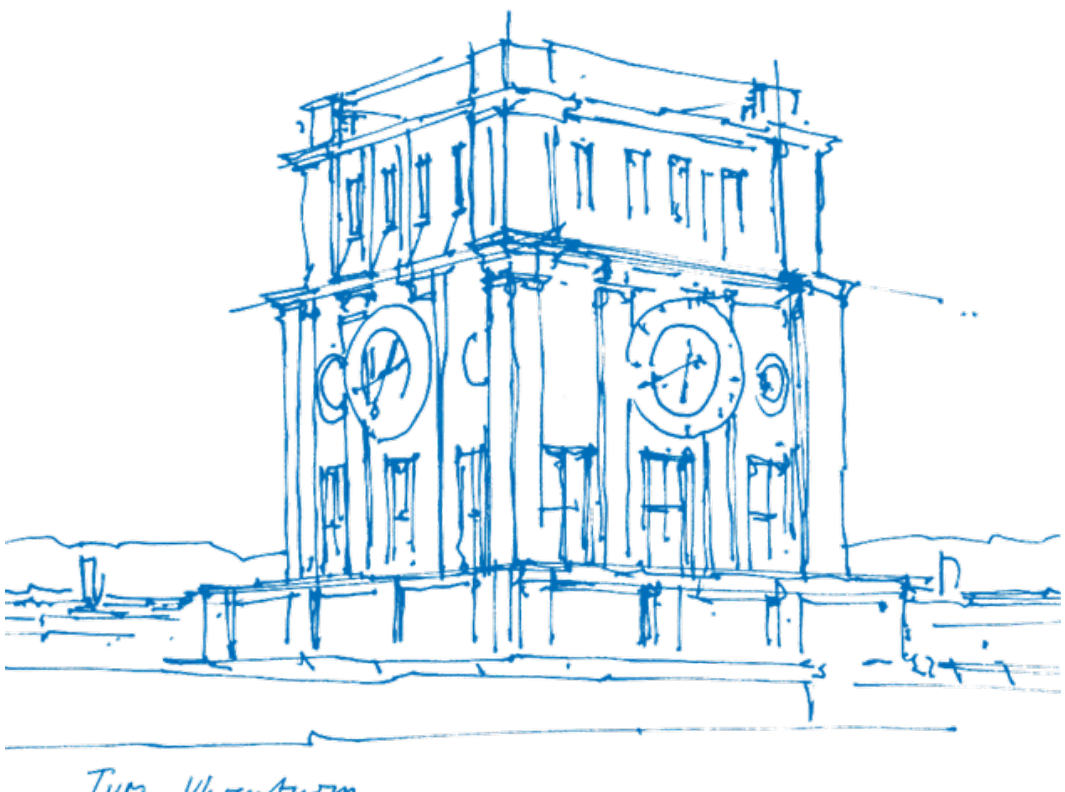
School of Computation, Information and Technology
Technical University of Munich

AACPP 2025

Week 7: Binary Search Trees



Mateusz Gienieczko, **Mykola Morozov**

School of Computation, Information and Technology
Technical University of Munich

2025.06.25

Fifth round – survey

Course Evaluation

Reminder – please fill out the Course Evaluation for our course!

Sixth round

Deadline – 01.07.2025, 10:00 AM.

Note – including this round **there will be only 5 more tasks**. Starting from next round there'll be only one task per week.

That gives 15 tasks total, 300 points to earn.

151 is guaranteed to pass.

We will decide the final grade thresholds based on the final ranking.

Remember – you can still submit tasks for *all* rounds until the end of semester (05.08.2025).

GRR – Garden Road Routine

Given a directed graph, find all vertices that can reach all other vertices in either G or G^{-1}.

In other words, $\{v \in V \mid \forall_{u \in V, u \neq v}. v \rightsquigarrow u \vee u \rightsquigarrow v\}$. Call such vertices *valid*.

Very easy brute-force solution – for each vertex check the condition with DFS and DFS on the inverted graph. $\mathcal{O}(nm)$

GRR – Garden Road Routine

Observation – nodes in a strongly connected component are equivalent, i.e.

- a node is valid if and only if all other nodes in its SCC are valid;
- if for any node u in the SCC some other node can reach it in either orientation (i.e. $v \rightsquigarrow u$ or $u \rightsquigarrow$ v), then it can also reach all nodes in u's SCC.

So we can fold all vertices into their SCCs and from now on assume a DAG.

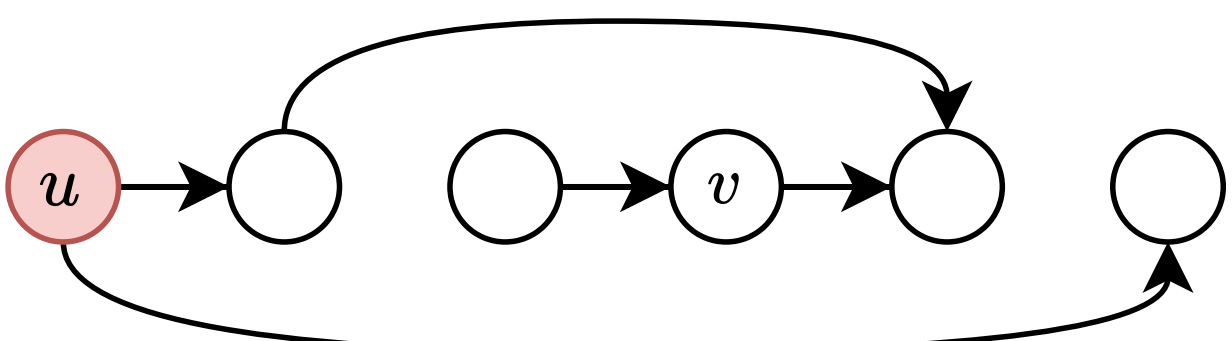
GRR – Garden Road Routine

It will be easier to consider when a node is *invalid*.

Since we have a DAG we can consider a topological order.

This simplifies, since if $u \prec v$ then $v \not\rightsquigarrow u$.

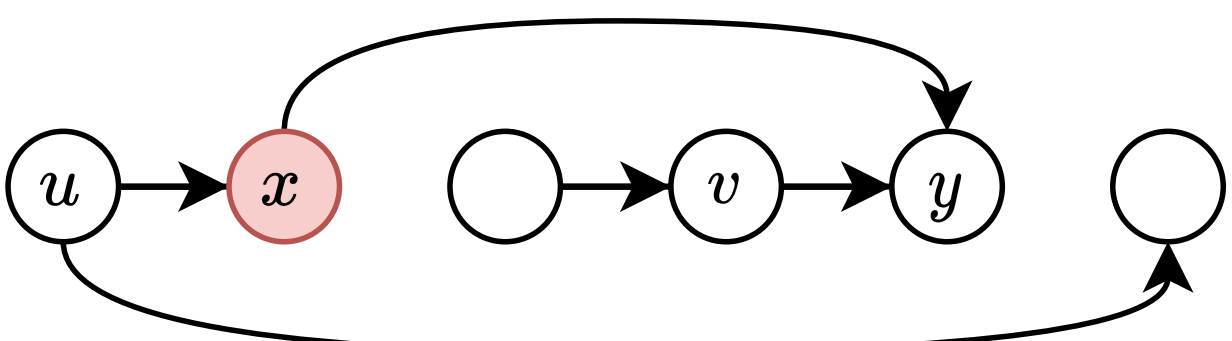
Now v is invalid if it is "jumped over" by any preceding node.



GRR – Garden Road Routine

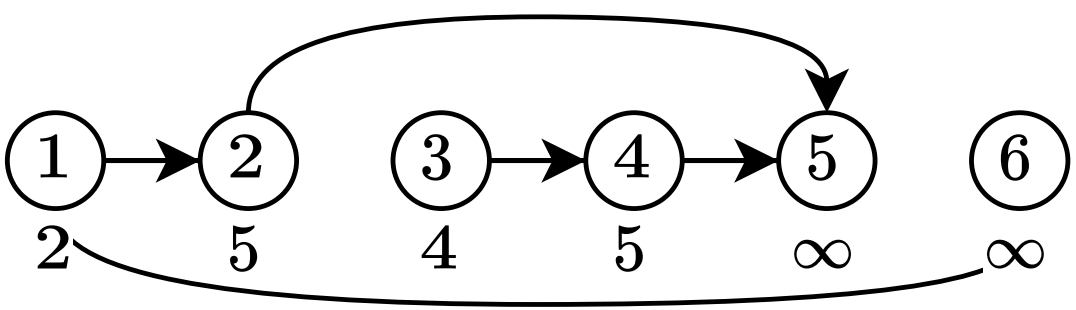
If it's jumped by *some* node then in particular there is a node x, $x \prec v$ such that the earliest vertex y it has a direct edge to satisfies $v \prec y$.

This makes the node definitely invalid. There might be more invalid nodes in the reversed graph, but we can handle that separately later.



GRR – Garden Road Routine

We can calculate the minimum reachable vertex and then examine the “holes” this creates.



Here, because 2 reaches only 5, both 3 and 4 are invalid. 6 is also invalid because $(5, \infty)$.

GRR – Garden Road Routine

Going from left to right we can find all invalid vertices by a kind-of two-pointers approach.

Keep v, u, where v is the current candidate that could still be invalid, and u is a node whose "hole" we are evaluating. Keep $u \prec v$.

If we have a hole (u, y) then all vertices in $[v, y)$ are invalid. Mark them and advance v accordingly.

GRR – Garden Road Routine

```
u = 0
v = 1
while u < n
  while v < minreach[u] and v < n
    invalid[v] = true
    v += 1
  u += 1
  if v <= u
    v = u + 1
```

GRR – Garden Road Routine

Run this again on reverse topo order and inverted edges to get all invalid vertices.

Everything we've done is in $\mathcal{O}(n + m)$.

LUC – Luck of the Paw

We can interpret the games as a *multigraph* on n vertices.

The task is then to find an orientation of edges (who wins which game) such that the maximal out-degree[1] over all vertices is minimal possible.

[1] Equivalently in-degree, no difference, here we'll assume (v, u) means v won against u.

LUC – Luck of the Paw

There is a relatively “simple” solution in $\mathcal{O}(m(m+n))$ that doesn’t use any fancy algorithms.

Idea: find any orientation of the edges and then repeatedly try to improve it.

LUC – Luck of the Paw

An improvement is a path $v \rightsquigarrow u$ such that $\text{outdeg}(v) \geq \text{outdeg}(u) + 2$.

If we invert all edges on this path we increase the outdeg of u by 1, decrease the outdeg of v by 1, and keep all other outdegs the same.

Algorithm: start with any orientation, find the vertex with max outdegree, find any path to a small-outdeg vertex, flip it.

LUC – Luck of the Paw

Perhaps surprisingly, this actually works, i.e. if we fix v of highest outdegree d and there is no path to a vertex with outdegree smaller by at least 2, then the minimal outdegree in any orientation is at least d.

Proof (sketch): Consider a DFS from v and count the number of vertices and edges in the DFS tree (all edges, including backward, forward, cross). Assume there is k vertices. Then there must be at least $(d-1)k+1$ edges, since the minimum outdeg is $d-1$ except for v which has one more. If we consider this subgraph, there can be no way of orienting the edges in a way that leads to $(d-1)$ max outdegree, since then we'd have at most $(d-1)k$ edges.

LUC – Luck of the Paw

What is the time complexity of this approach?

The search for a path is in $\mathcal{O}(n + m)$.

It can be proven that if a given vertex is ever selected as the start of the path, then it will never be selected as the endpoint of a path and thus its outdegree will never *increase*.

Since each vertex can only be maximal as many times as its degree in the graph, and sum of degrees is $2m$, we have $\mathcal{O}(m)$ iterations.

In total we have $\mathcal{O}(m(n + m))$, which should yield us 6 points.

LUC – Luck of the Paw

There is another bound on the number of iterations, namely the outdegree of the initial orientation minus the end result.

By choosing a reasonable start orientation and being efficient in the implementation of all steps it was possible to get 8 points[2].

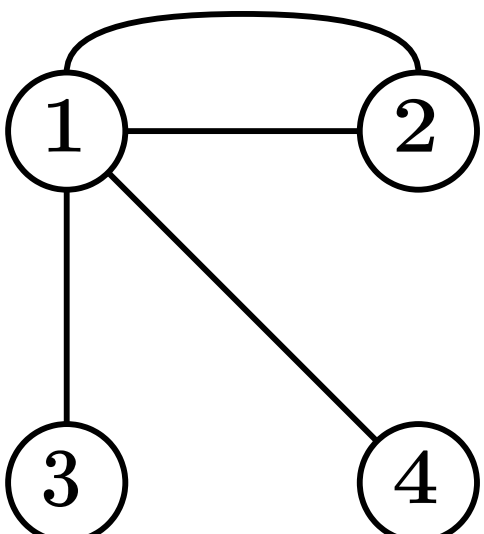
[2] Or even 10, but that's because our time limits were a bit too generous.

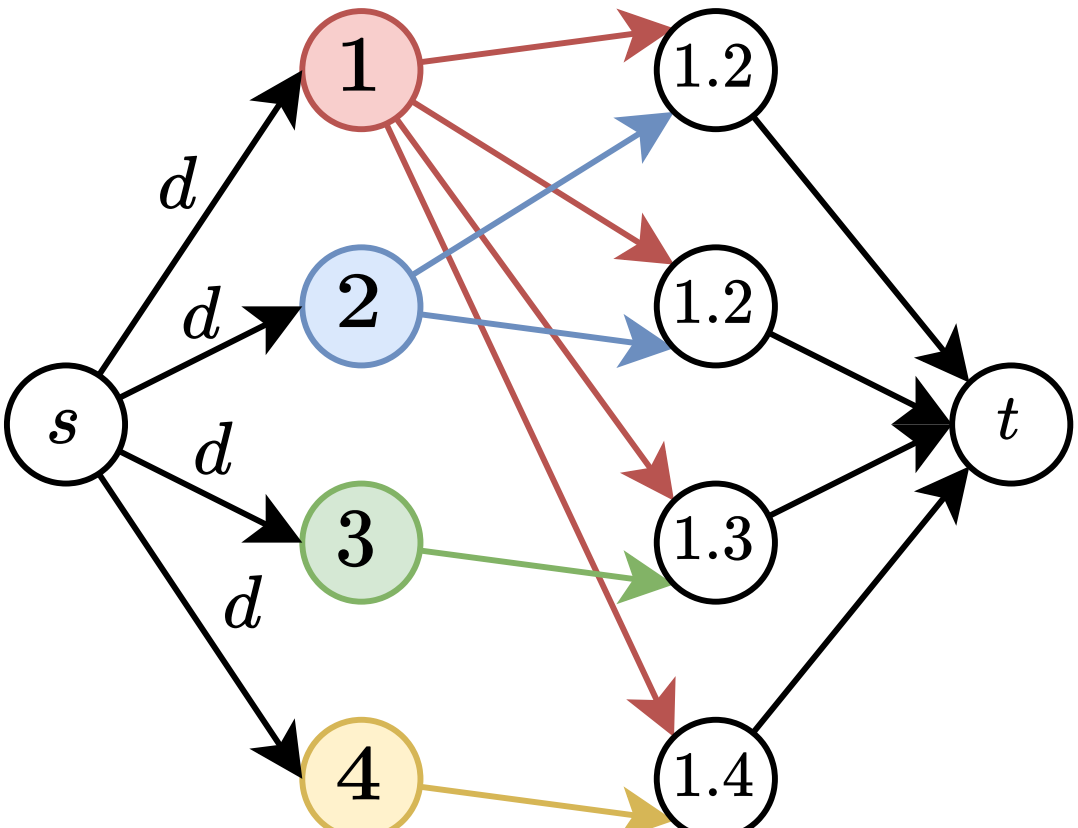
LUC – Luck of the Paw

Let's try a different approach – the idea of finding augmenting paths looks awfully similar to a flow problem.

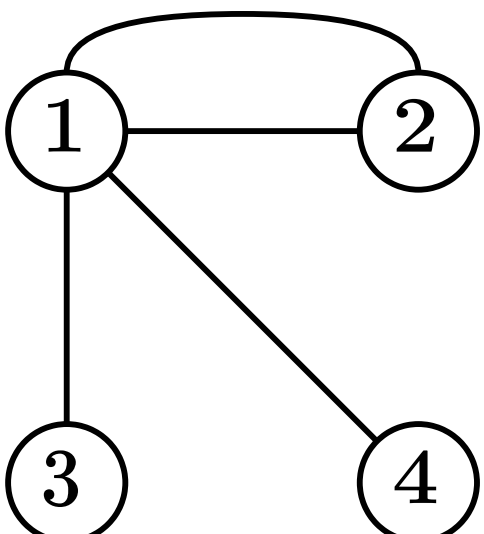
We can design a flow network in which there exists a flow of size m if and only if the graph can be oriented such that the max outdegree is at most d.

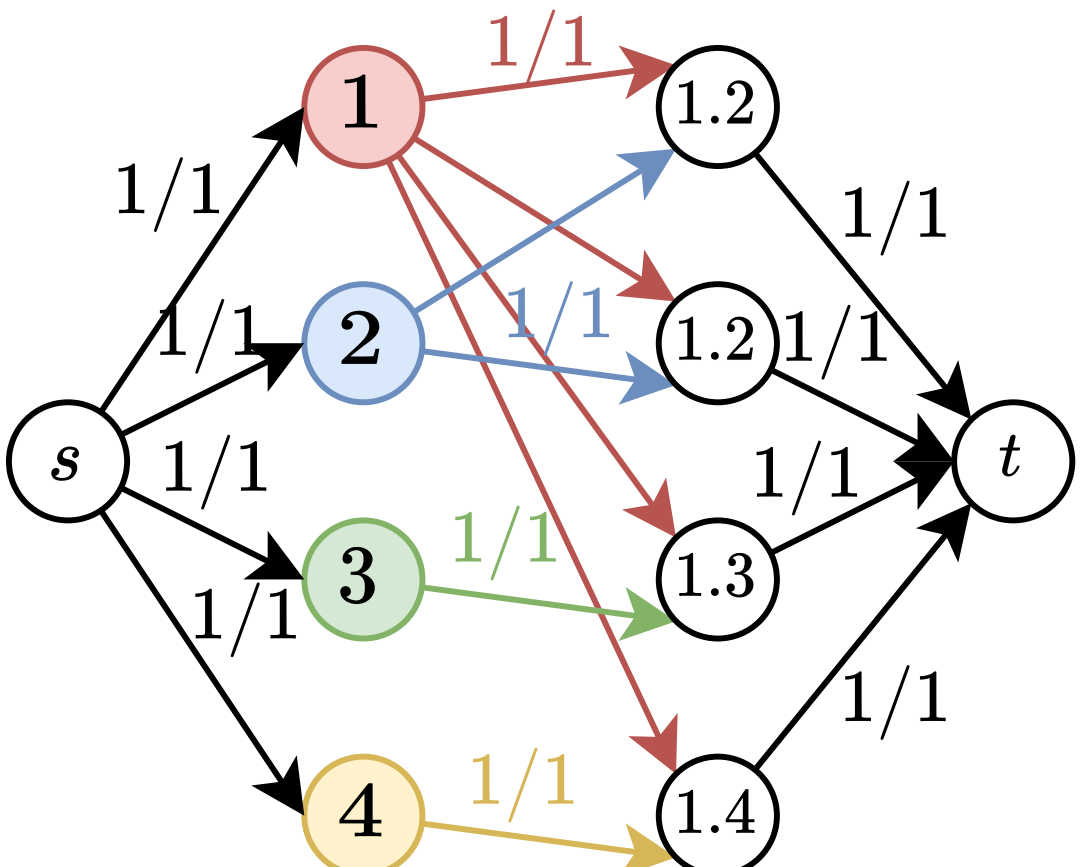
LUC – Luck of the Paw



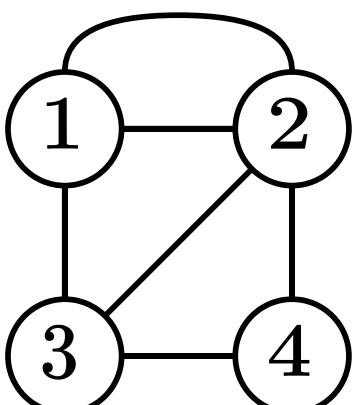


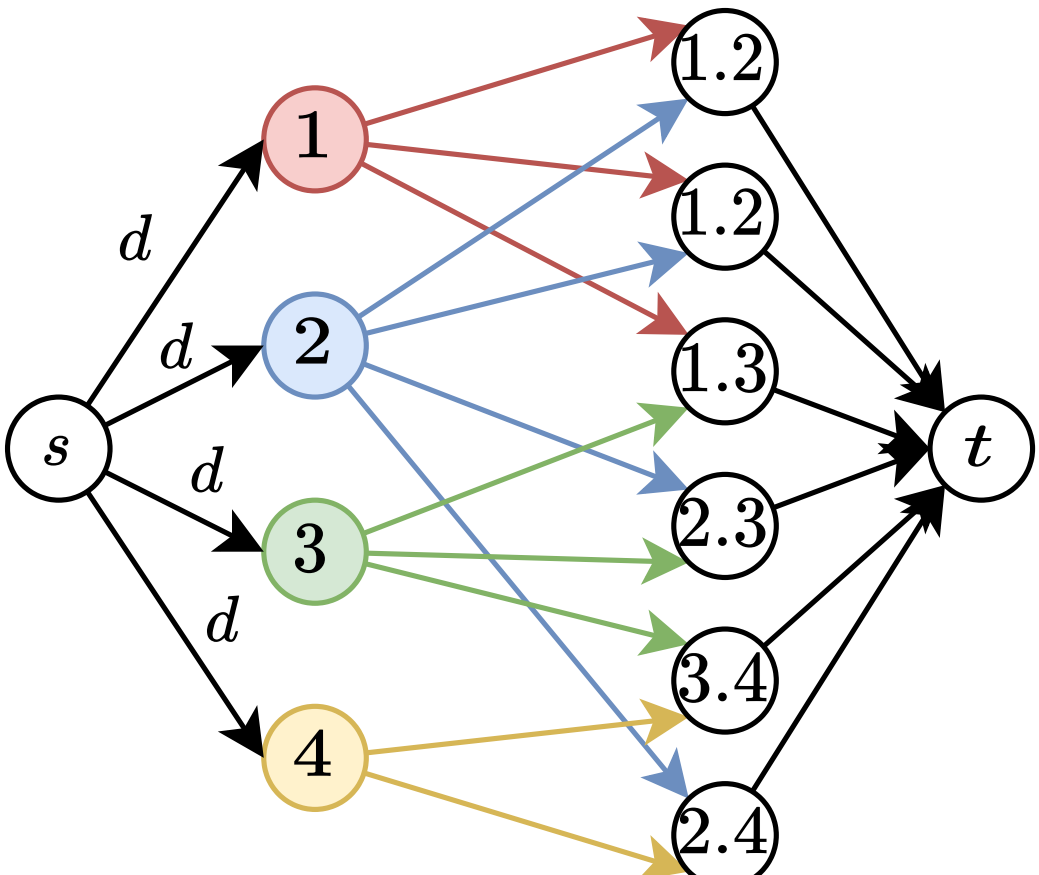
LUC – Luck of the Paw



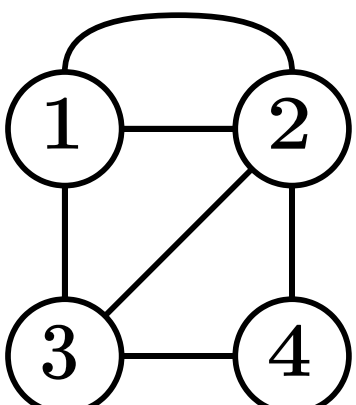


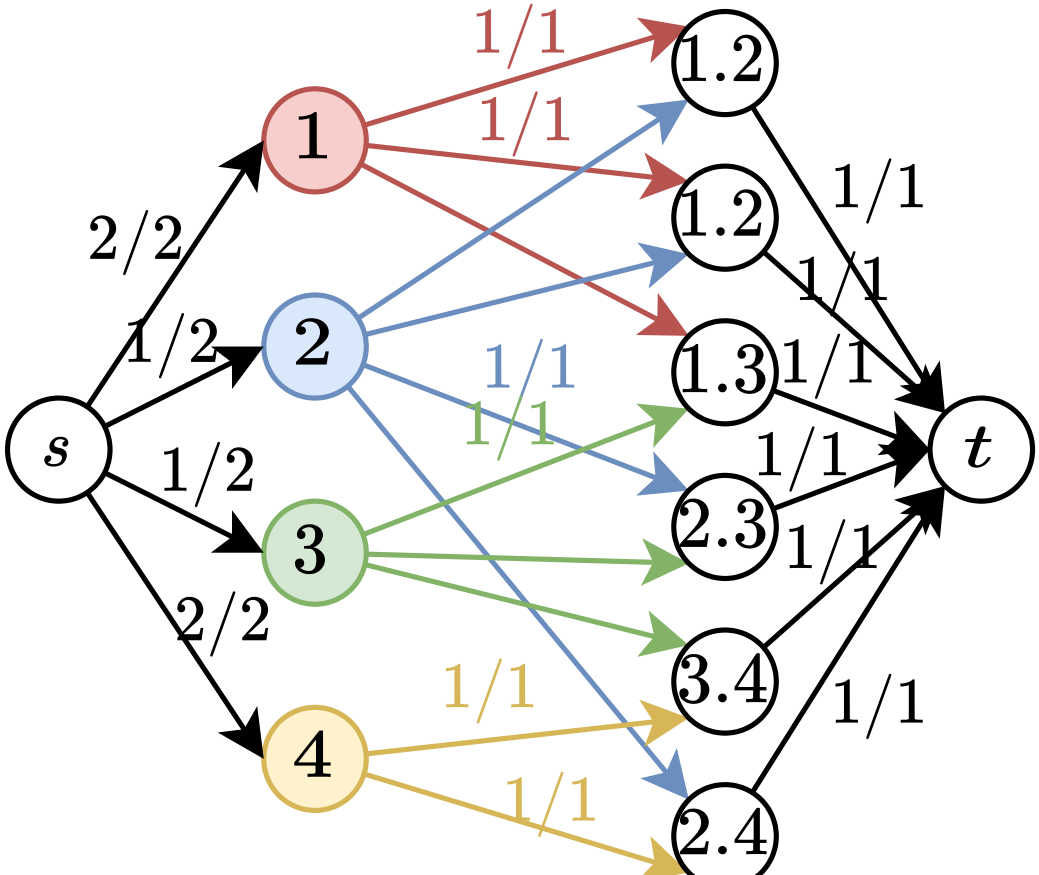
LUC – Luck of the Paw





LUC – Luck of the Paw





LUC – Luck of the Paw

Algorithm: binary-search the result. For a given d construct the flow network and find the max-flow. If m, then we have an upper-bound, if not, we have a lower-bound.

Restoring the result is easy – look at the edges from vertex-vertices to edge-vertices and orient them according to the flow.

LUC – Luck of the Paw

The max result can be $\frac{m}{n}$. So we call max-flow $\mathcal{O}(\log m)$ times.

What algorithm to use for max-flow and what is the runtime?

LUC – Luck of the Paw

In the flow network there are $n + m + 2 = \mathcal{O}(m)$ vertices and $n + 3m = \mathcal{O}(m)$ edges.

The maximum flow can be of the order $\mathcal{O}(m)$.

The bound from Ford-Fulkerson is $\mathcal{O}\left(m^2\right)$ which is as good as the previous solution (but is slower in practice).

Normal analysis of Dinitz is no better, but a careful analysis gives a better bound.

LUC – Luck of the Paw

Each augmenting path in the graph is essentially as if we were computing a half-matching, where vertices on the left can be matched to more than one.

Every path in a blocking flow is alternating, it starts from a vertex on the left that has not yet matched d vertices, alternates picked and unpicked edges in the middle, and finishes with a yet-unmatched vertex on the right.

This gives us the same analysis as for Hopcroft-Karp.

LUC – Luck of the Paw

After k blocking flows the length of the shortest augmenting path is at least $k + 1$.

After $\sqrt{m} - 1$ blocking flows the shortest augmenting path is $\sqrt{m}$.

Because there's at most $\sqrt{m}$ disjoint paths of length $\sqrt{m}$ that is the maximum number of additional phases. So the number of total phases is $\mathcal{O}\left(\sqrt{m}\right)$.

We get $\mathcal{O}\left(m\sqrt{m}\right)$.

Combined with the binary search we have $\mathcal{O}\left(m\sqrt{m}\log(m)\right)$.

LUC – Luck of the Paw

After k blocking flows the length of the shortest augmenting path is at least $k + 1$.

After $\sqrt{m} - 1$ blocking flows the shortest augmenting path is $\sqrt{m}$.

Because there's at most $\sqrt{m}$ disjoint paths of length $\sqrt{m}$ that is the maximum number of additional phases. So the number of total phases is $\mathcal{O}\left(\sqrt{m}\right)$.

We get $\mathcal{O}\left(m\sqrt{m}\right)$.

Combined with the binary search we have $\mathcal{O}\left(m\sqrt{m}\log(m)\right)$.

And this is still too slow!

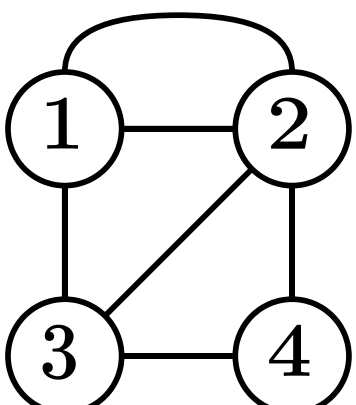
LUC – Luck of the Paw

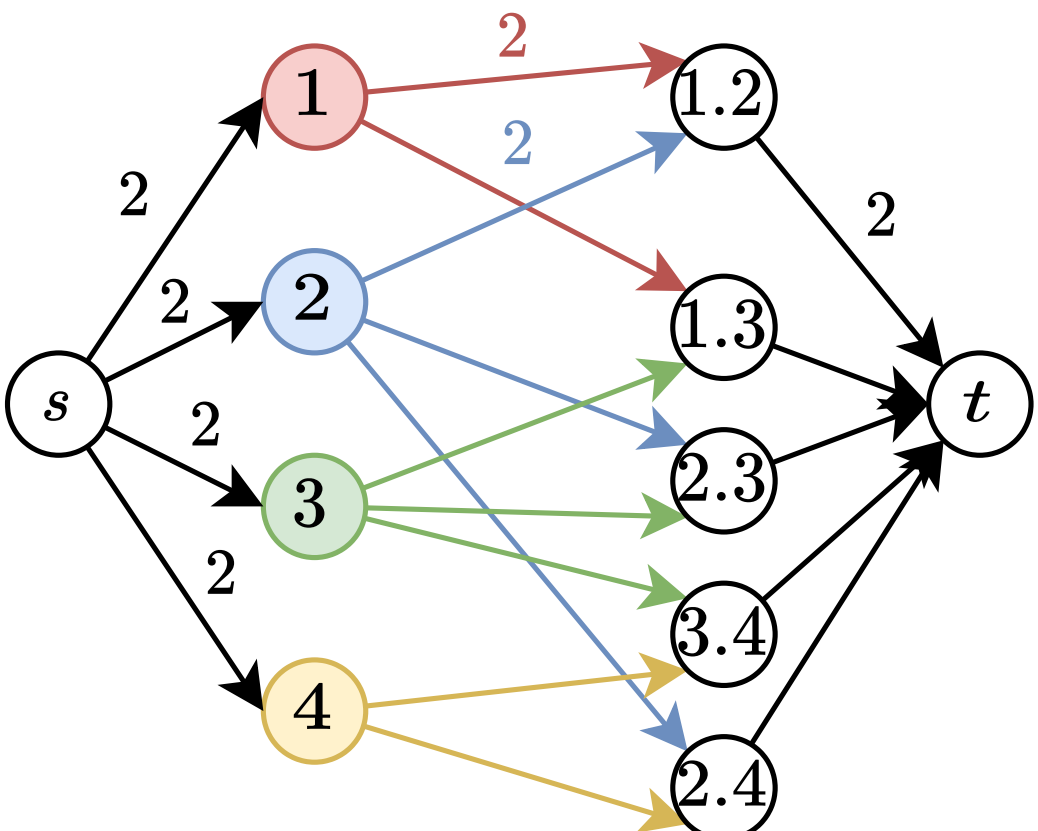
That flow is sufficient for many graphs, but not if the solution really is close to m.

An example of such a graph is an k-times-clique, i.e. we take a complete graph and duplicate all the edges k times. There are $k\frac{n(n-1)}{2}$ edges in such a graph and the best orientation is $\frac{k(n-1)}{2}$.

Considering duplicates of the same edge is actually wasteful...

LUC – Luck of the Paw





LUC – Luck of the Paw

We can now also bound the size of the network by $\mathcal{O}(n^2)$.

Since the size of the result is larger the denser the graph is, this is a meaningful improvement.

E.g. a clique on 316 vertices has almost 10^5 edges, yet the result is bounded by 158.

A clique on 79 vertices where each edge is duplicated 16 times has a result bounded by 640, but the entire flow network has only 6 214 vertices.

So we have $\mathcal{O}(n^3)$ when the graph is dense.

The previous bound of $\mathcal{O}(m\sqrt{m})$ also holds, and in a sparse graph $\frac{m}{n}$ is very low.

Recall the plan

- Greedy and dynamic programming (DP)
- Trees
- Graphs
- Ways to turn graphs into trees (DFS, BFS, Dijkstra, MST)
- Ways to run DP on graphs (Toposort)
- Advanced graph algorithms (Matchings, flows)
- **Binary Search Trees** ← *we are here*
- Number theory
- String algorithms (KMP, tries, suffix tables)
- Some problems can't* even be solved efficiently (NP-completeness)

BST – General idea

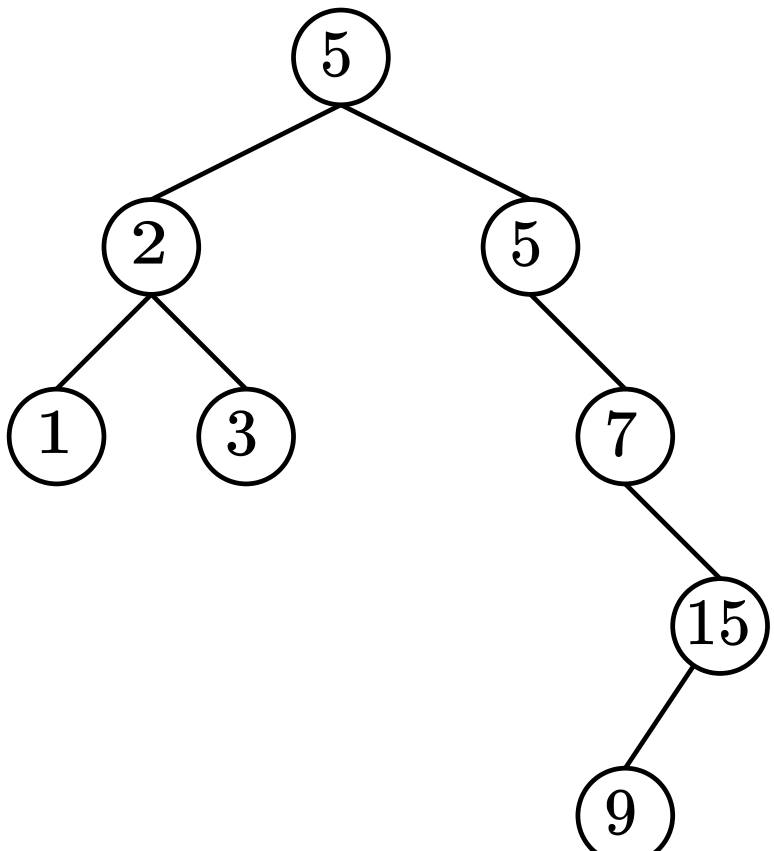
We want to store keys ordered by some total order.

Organise them into a binary tree where, if a node has key k, then every key in the left subtree is $< k$ and every key in the right subtree is $\geq k$.

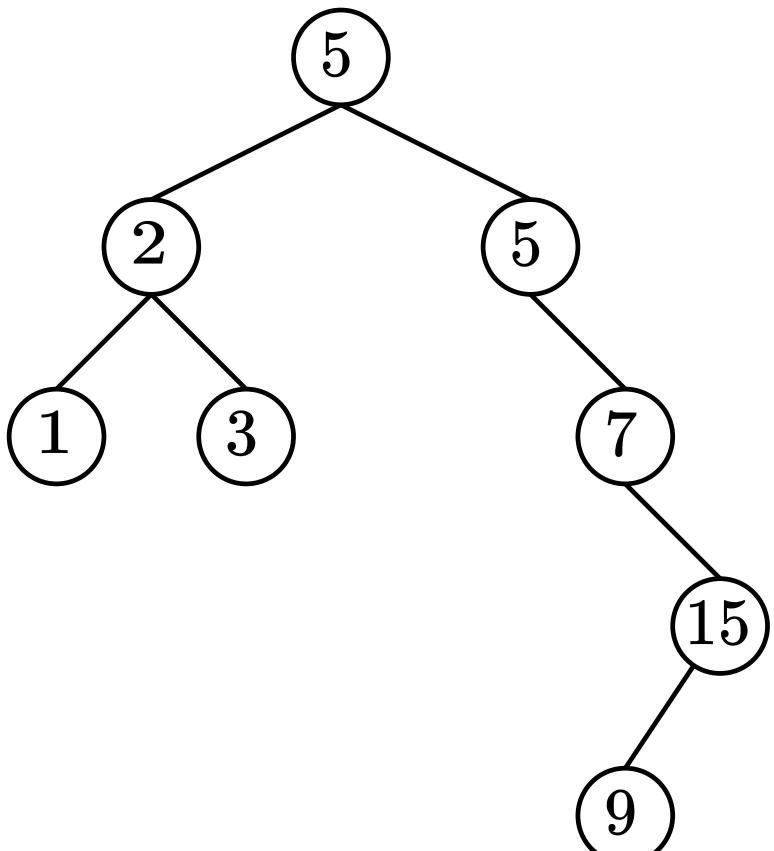
Searching for a key we always know where to go.

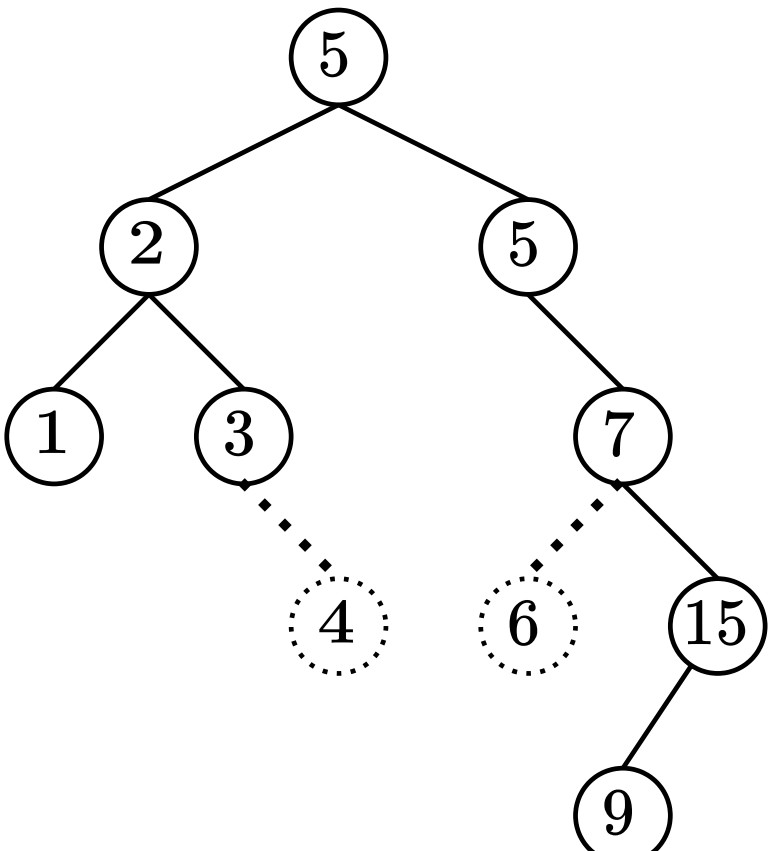
Search time is $\mathcal{O}(\text{depth})$.

BST – General idea



BST – General idea





BST – Lookup

```
fn lookup(node, key)
  if node.key == key
    return node
  else if node.key <= key
    if node.right is None { return None }
    else
      return lookup(node.right, key)
  else
    if node.left is None { return None }
    else
      return lookup(node.left, key)
```

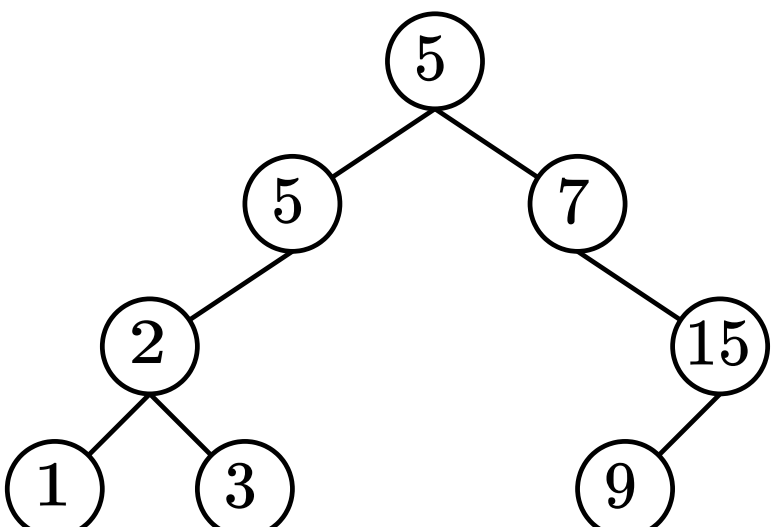
BBST

A *balanced* binary search tree is one where the depth is $\mathcal{O}(\log n)$ for n keys.

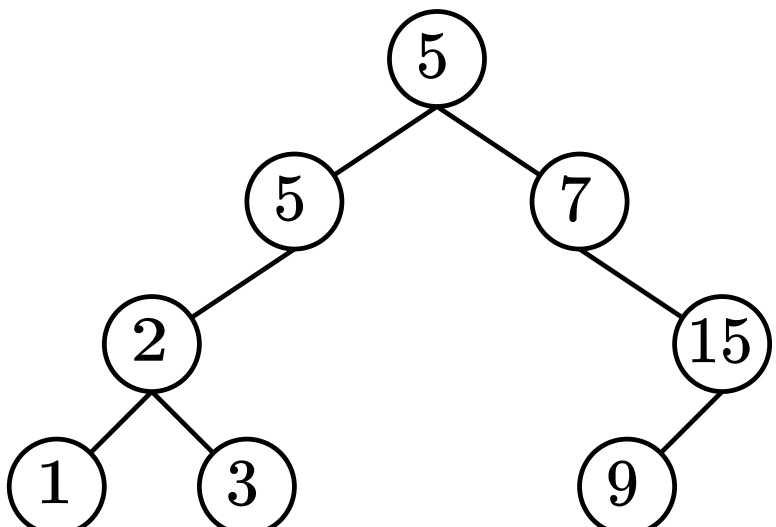
The search time is always logarithmic then.

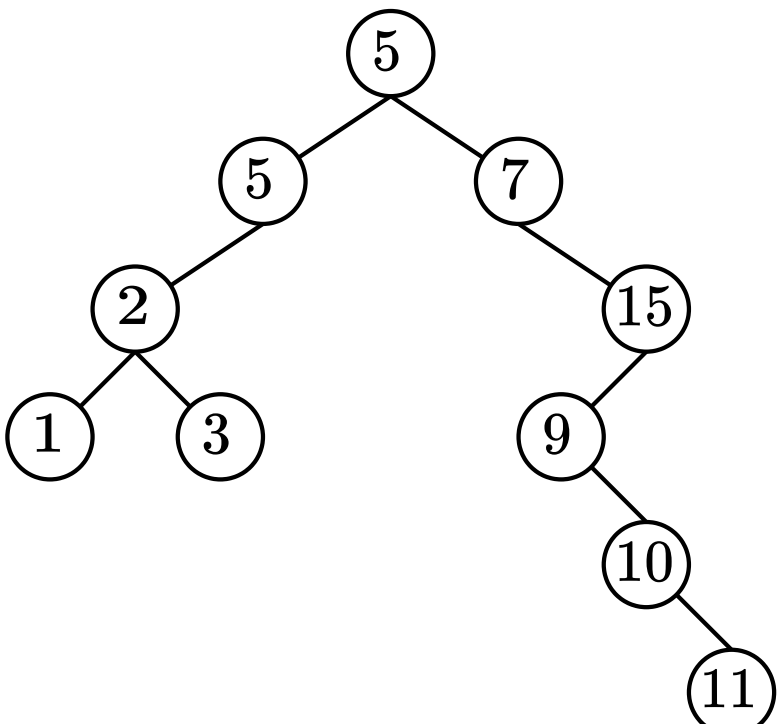
The challenge is how to perform key insertion to maintain the balance.

BBST



BBST





BST – AVL

Adelson-Velsky and Landis trees maintain an invariant the the difference between the heights of two siblings is at most one.

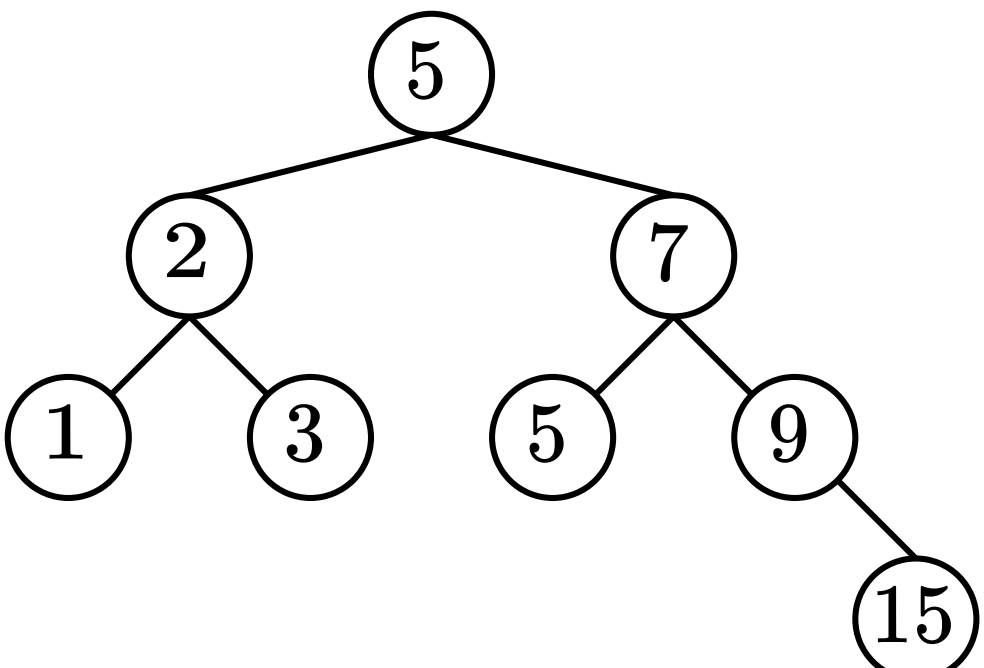
The tree needs to maintain the height difference, which takes 2 bits of memory per node.

Rebalancing is done via tree rotations.

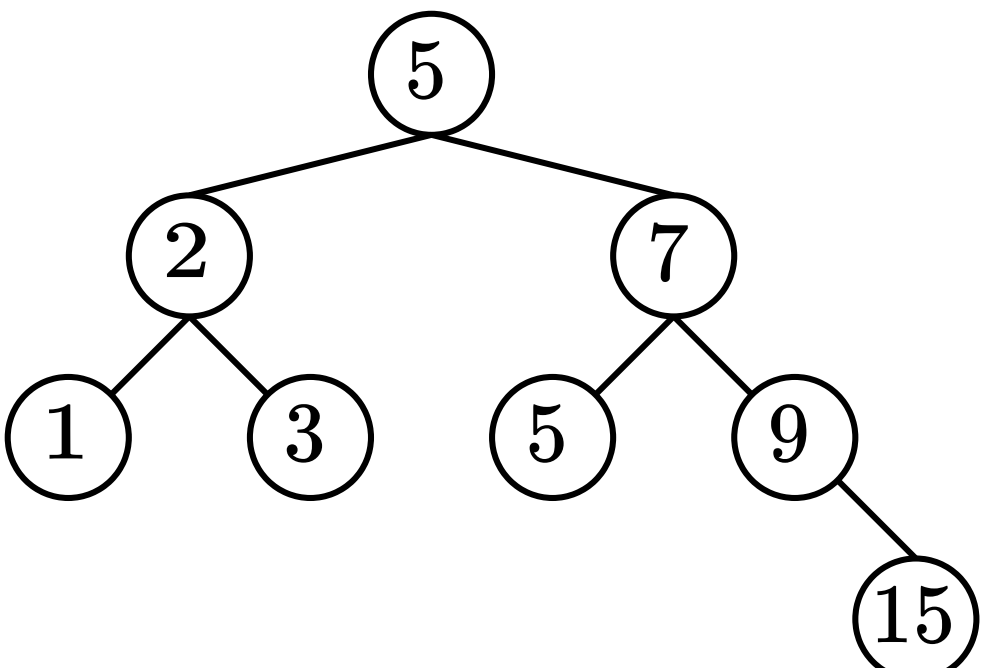
All operations are $\mathcal{O}(\log n)$ (a tree of height h has at least F_{h+2} - 1 nodes[3]).

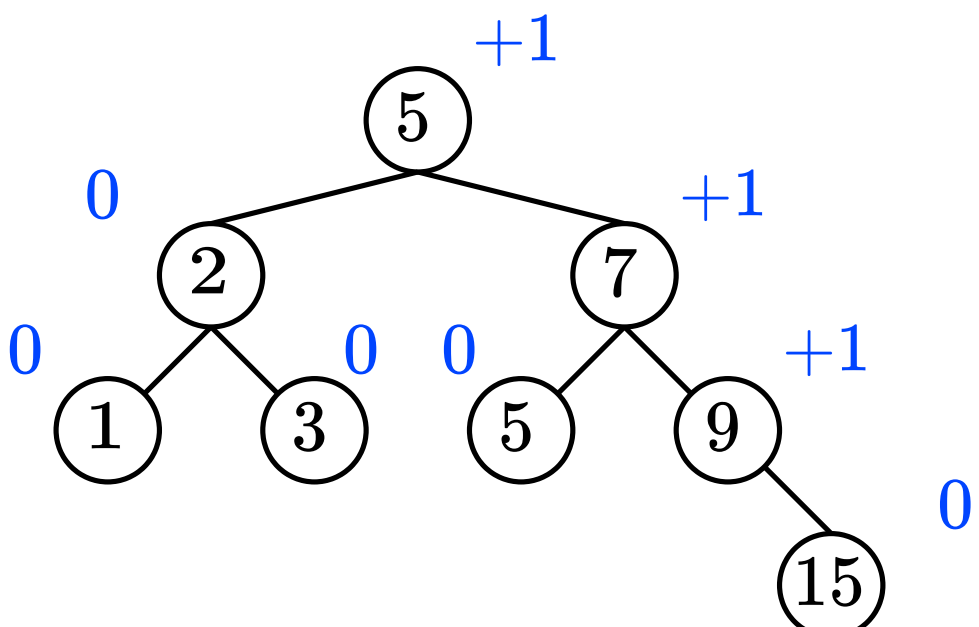
[3] F_n is the n-th Fibonacci number.

BST – AVL

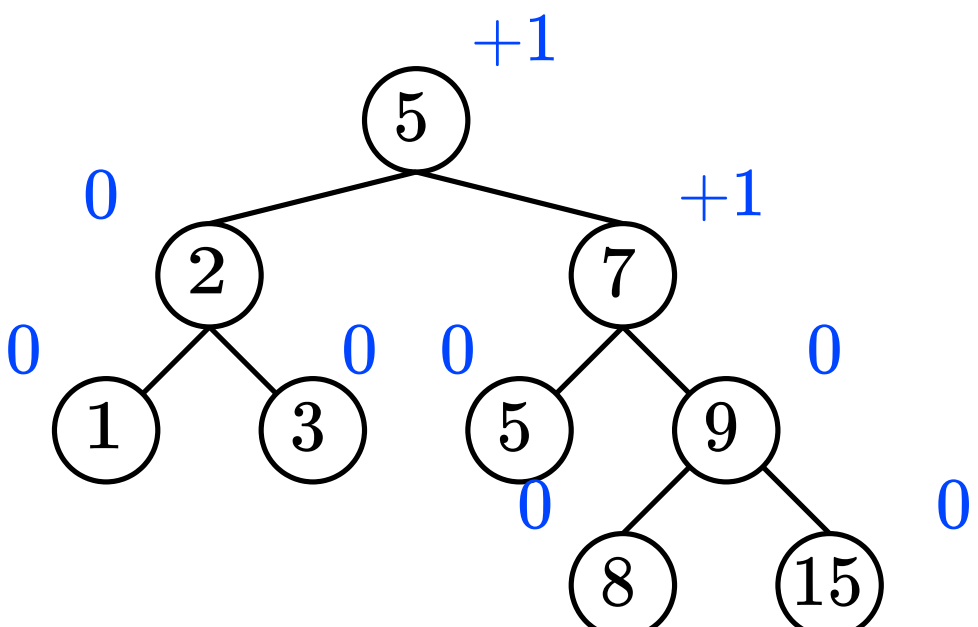


BST – AVL

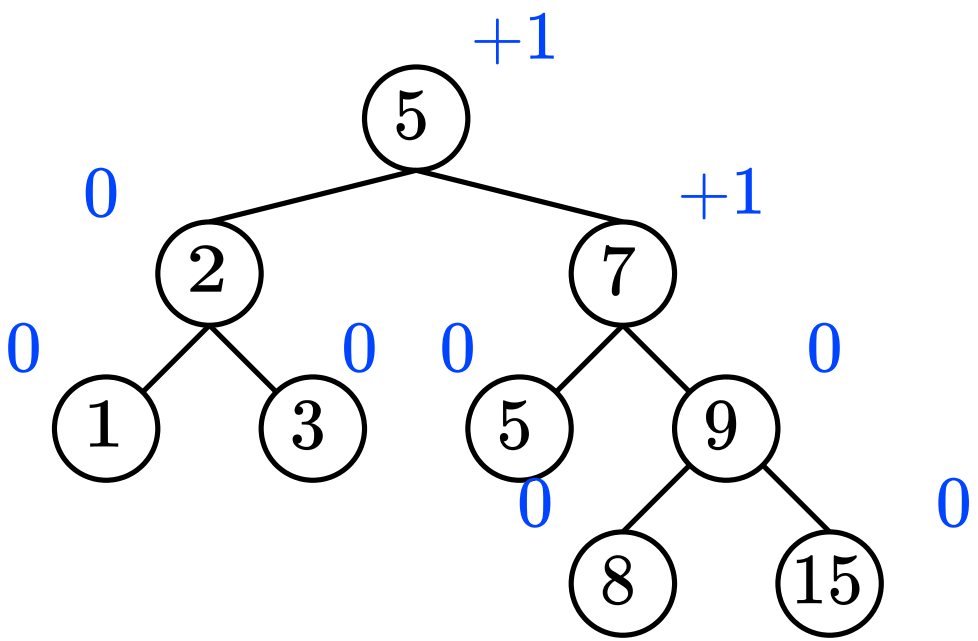


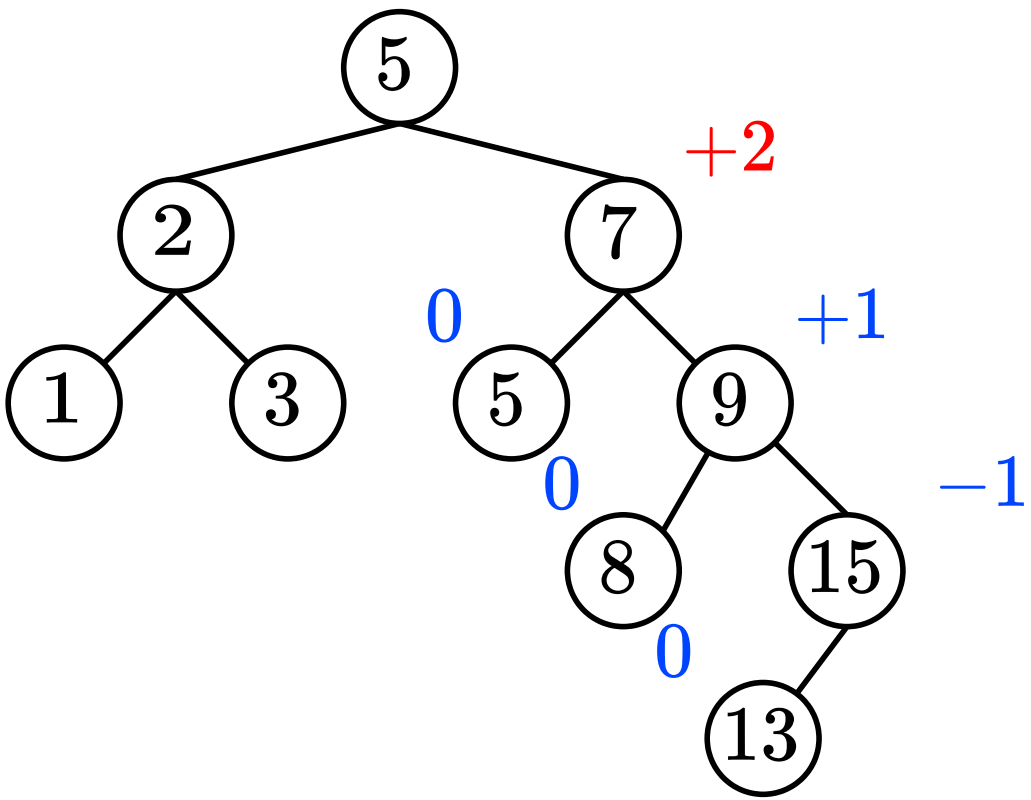


BST – AVL insertion

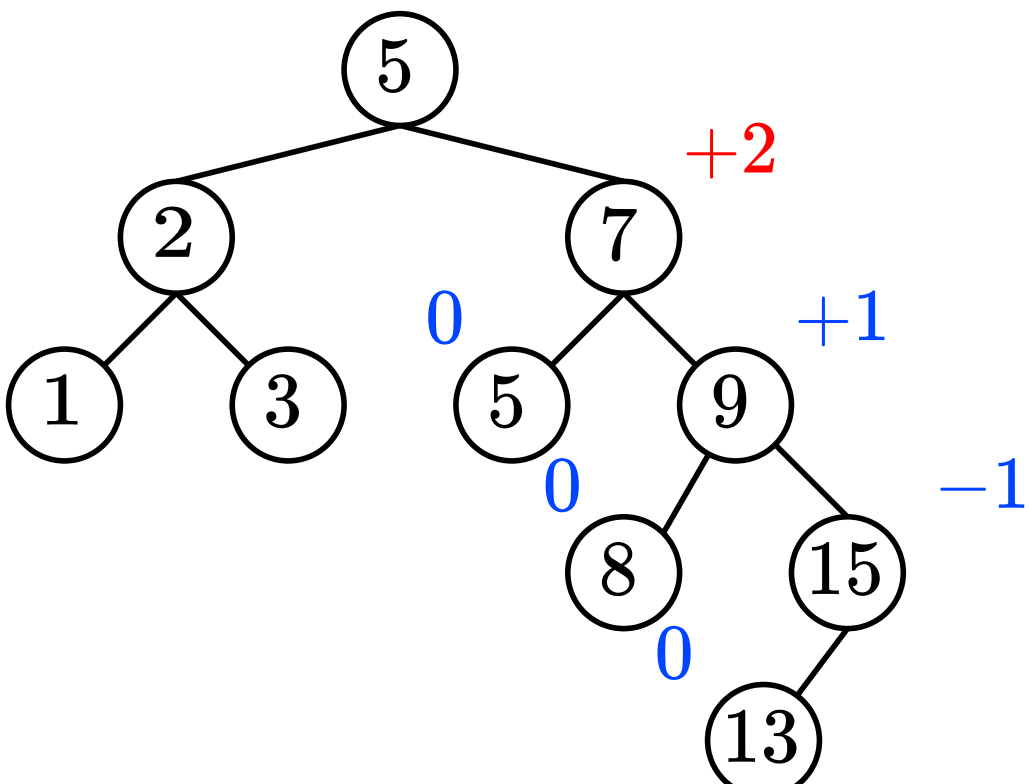


BST – AVL insertion

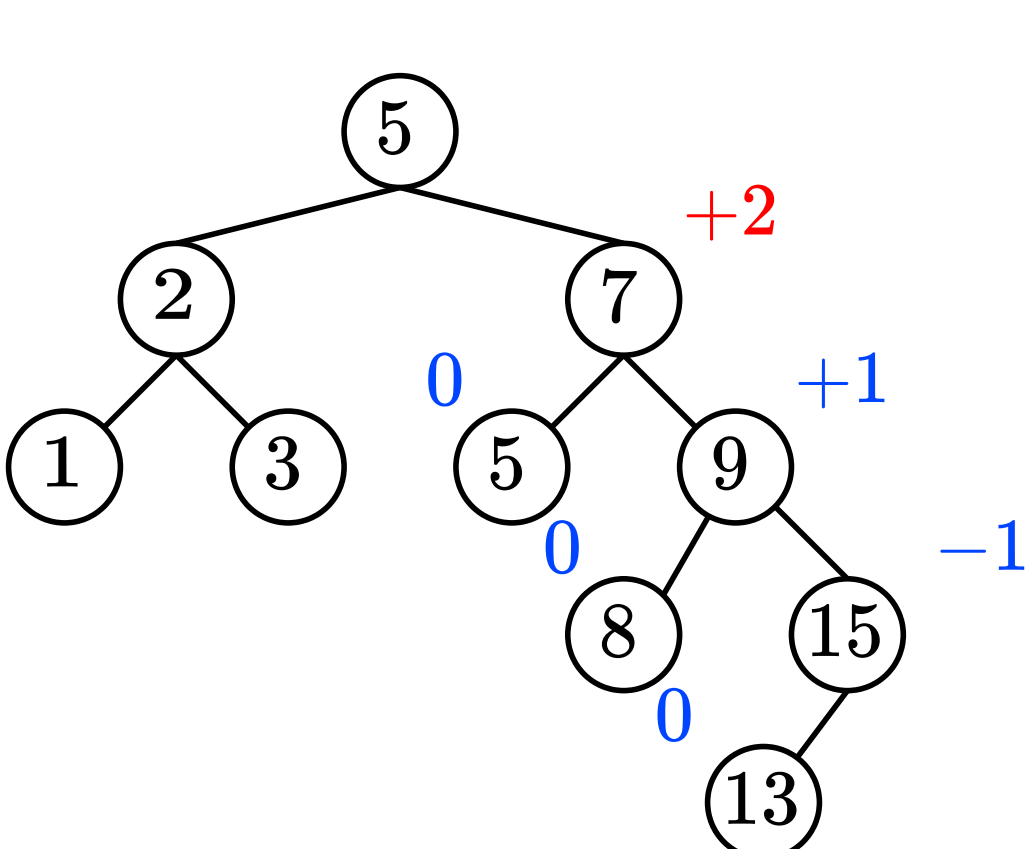


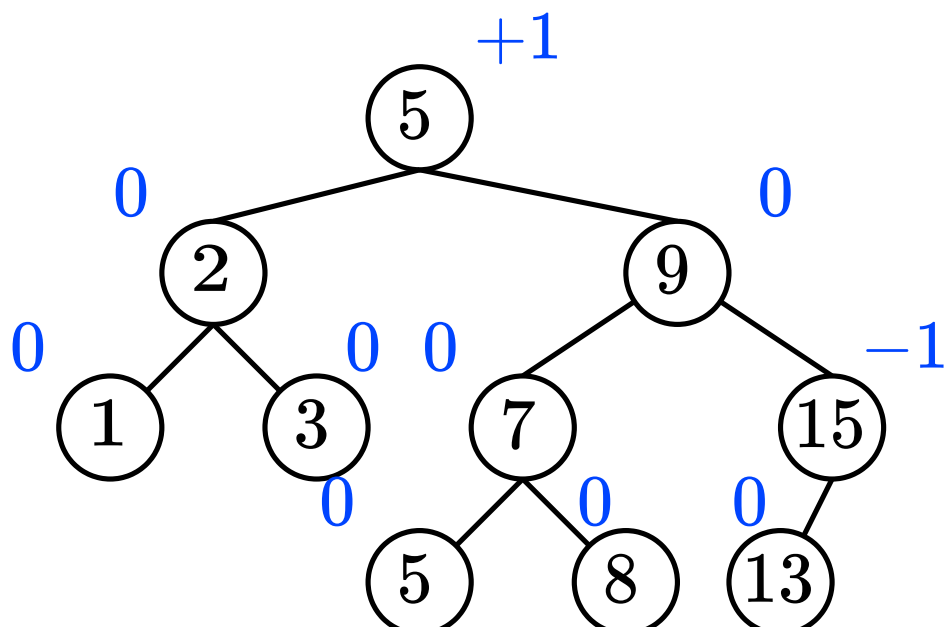


BST – AVL balancing (left rotation)



BST – AVL balancing (left rotation)





BST – Red-Black Tree

Red-black trees are memory-efficient BSTs that require only one bit to specify colour of a node – red or black.[4]

Balance is maintained with a couple of invariants, rotations and recolourings:

- A null node is black;
- A red node does not have a red child
- Every path from a node to any of its leaf nodes goes through the same number of black nodes.

[4] Interesting fact – RB trees are inspired by 2-3-4 trees invented by Rudolf Bayer. Prof. Bayer is a professor emeritus at our database chair 😄 He also co-invented B-trees.

BST – Red-Black Tree

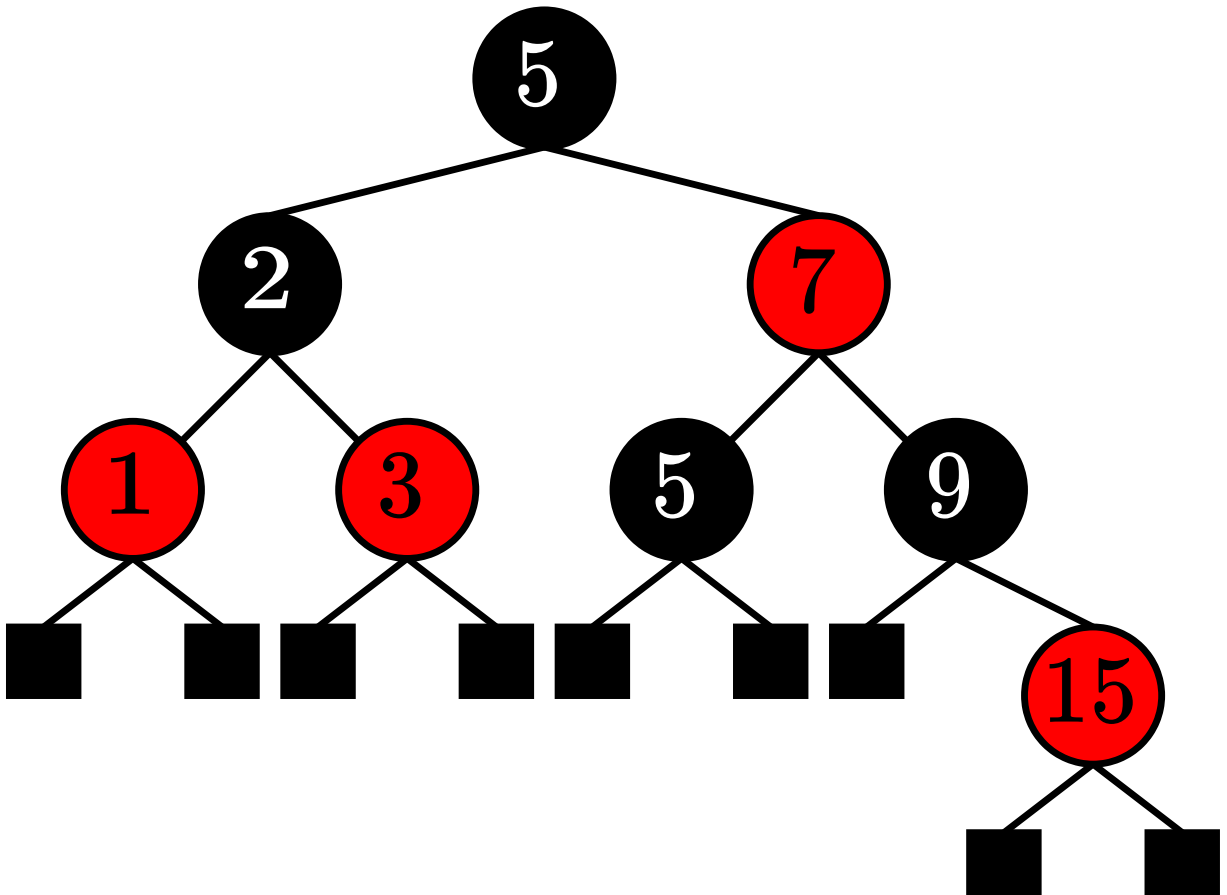
Based on the invariants, the path to the deepest leaf is at most two times the path to any leaf.

More precisely, the *black-height* – height counting only black nodes – is at least half of the total height.

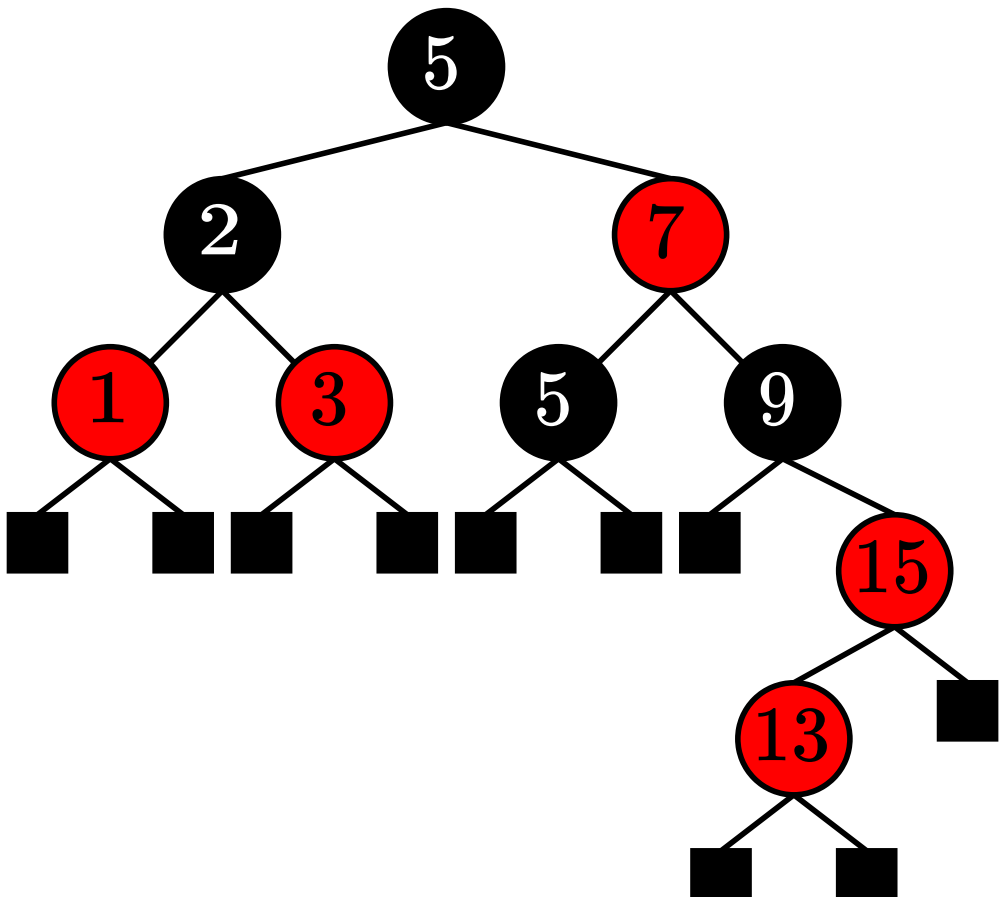
We get $\mathcal{O}(\log n)$ operations.

RB trees are on average deeper and slower than AVLs, but they require less memory (1 bit per node instead of 2 bits).

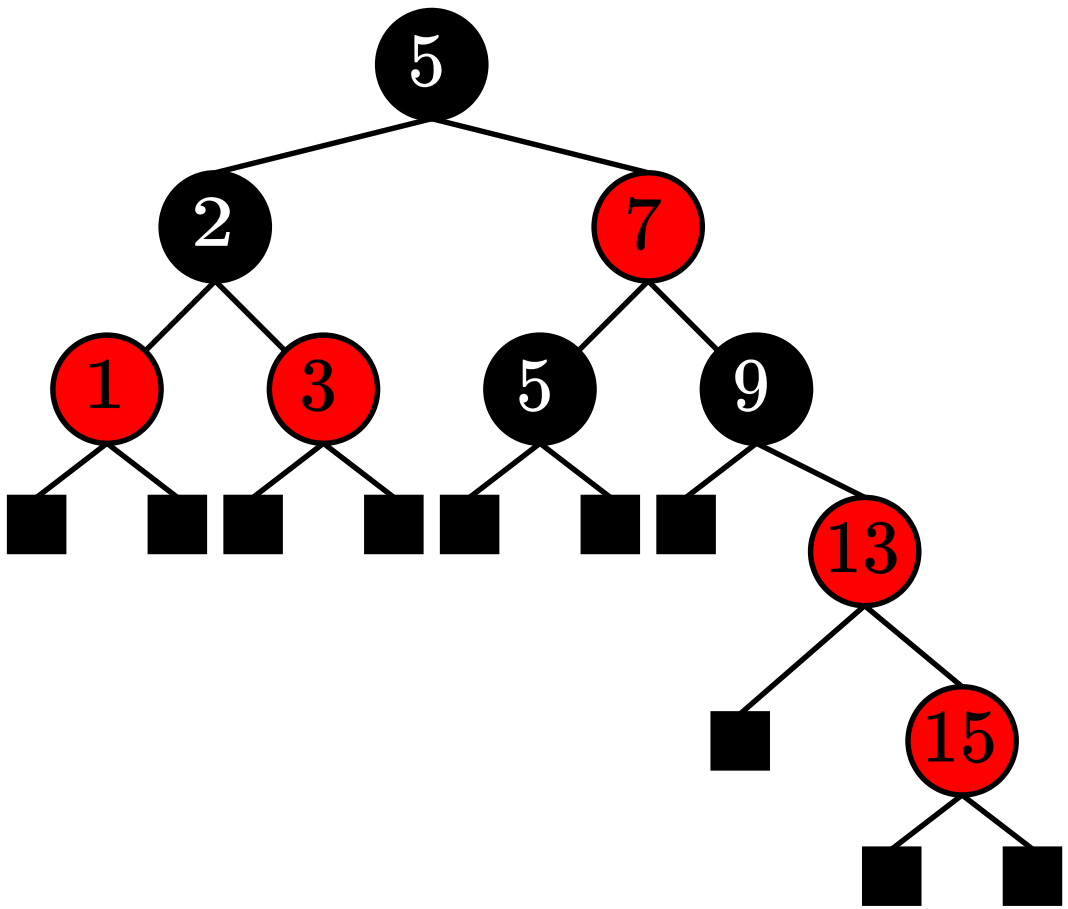
BST – Red-Black Tree



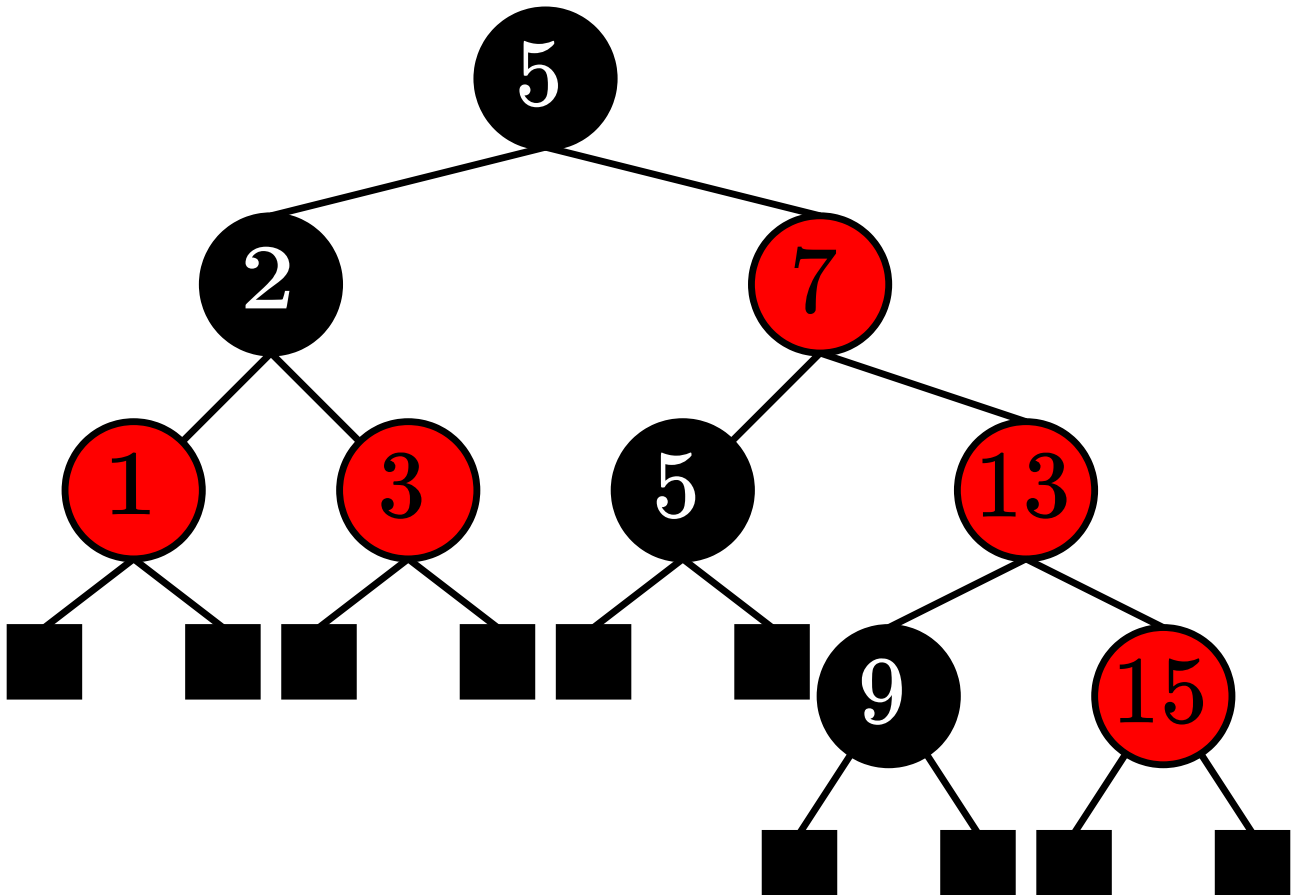
BST – Red-Black Tree (red violation)



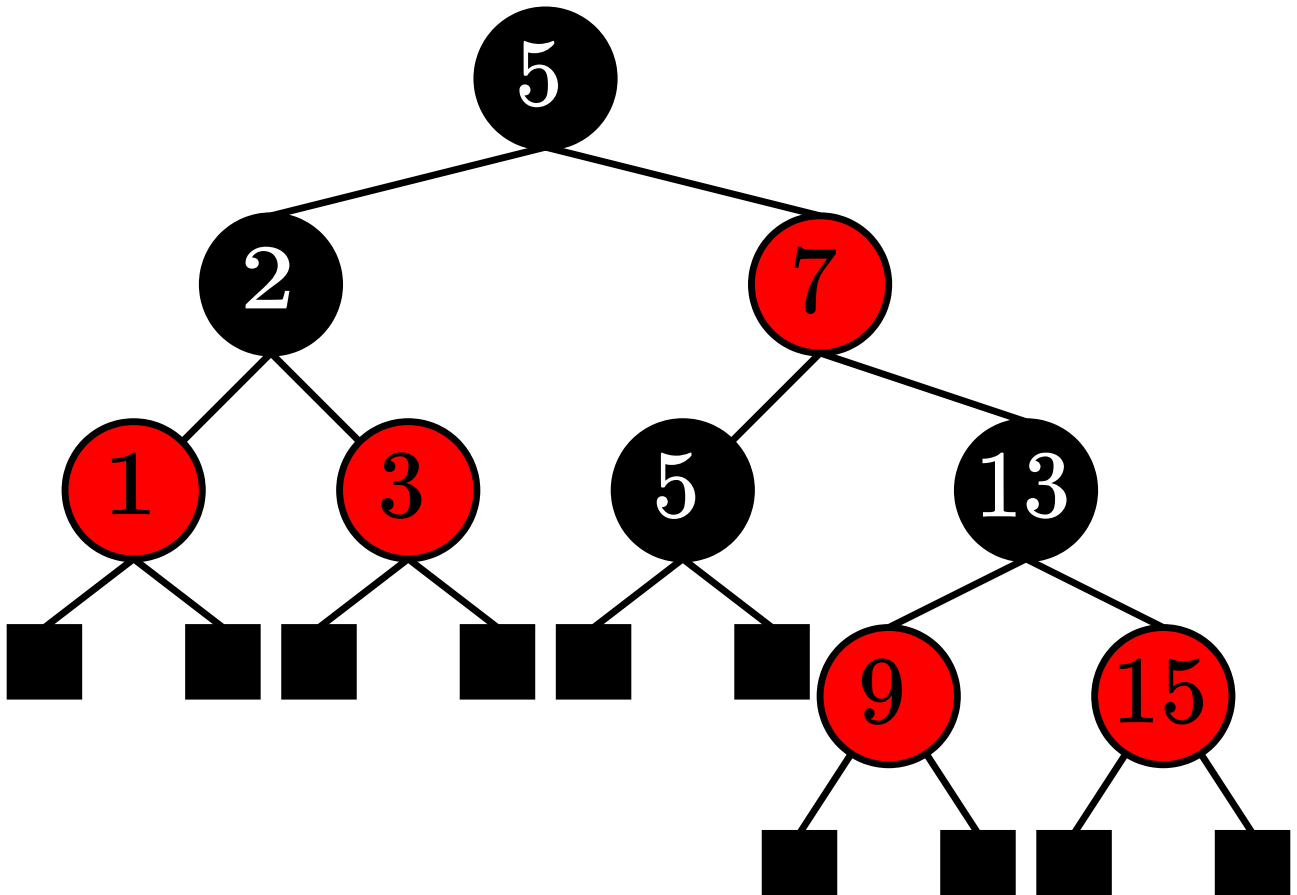
BST – Red-Black Tree (red violation)



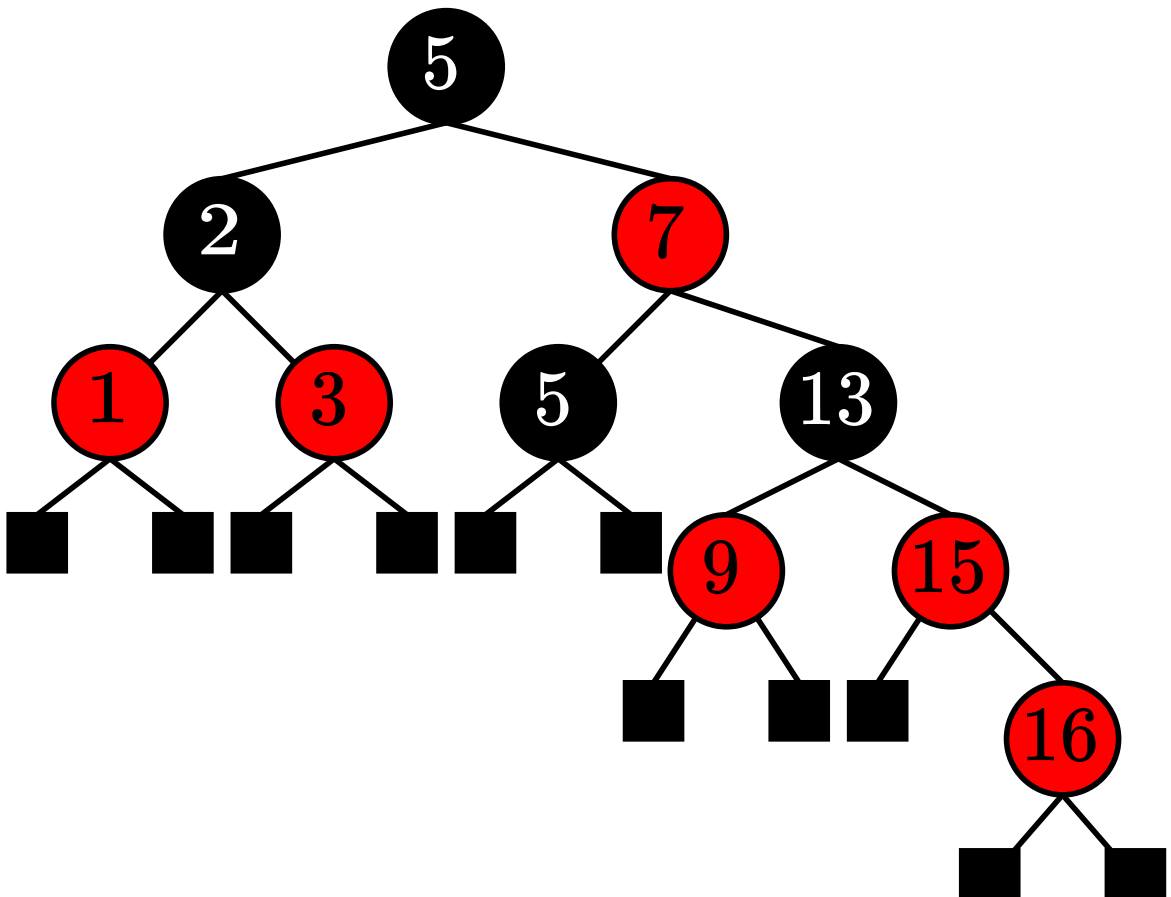
BST – Red-Black Tree (red violation)



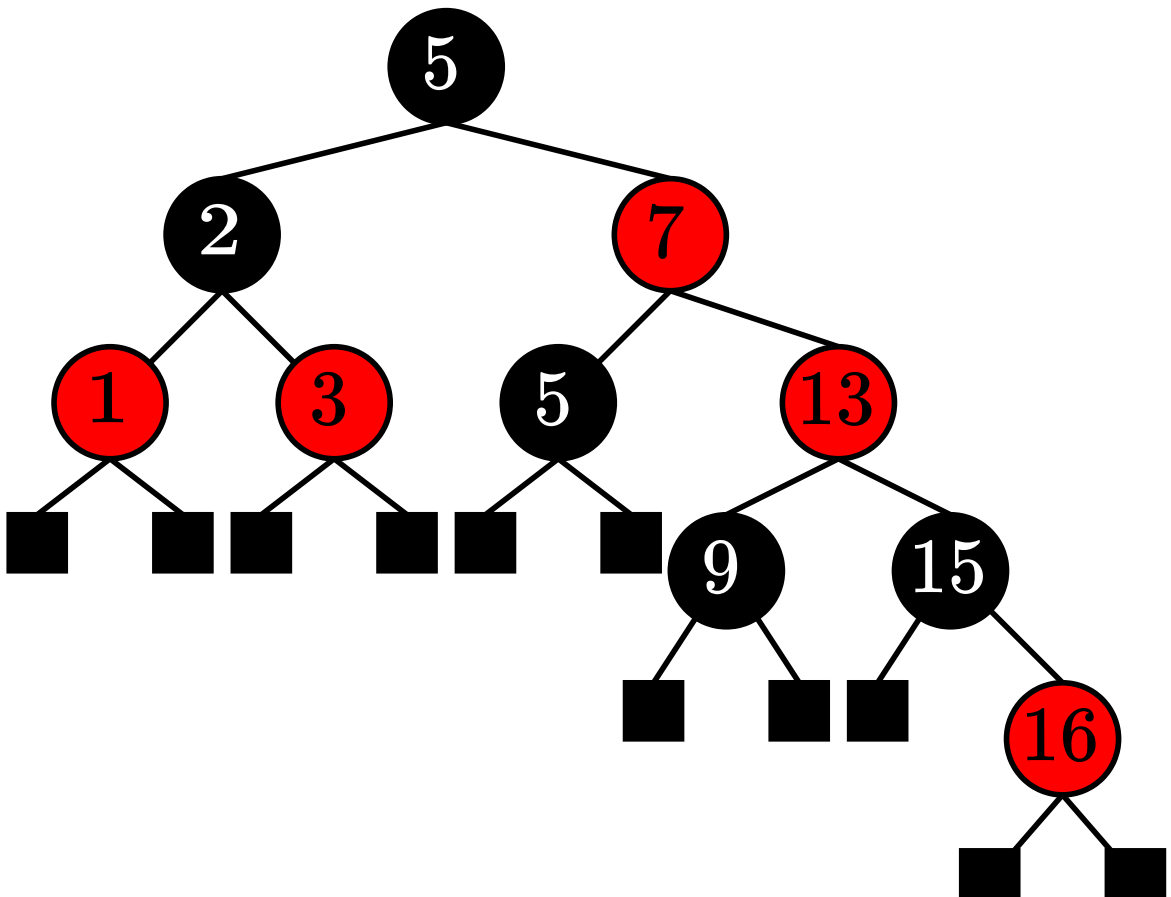
BST – Red-Black Tree (red violation)



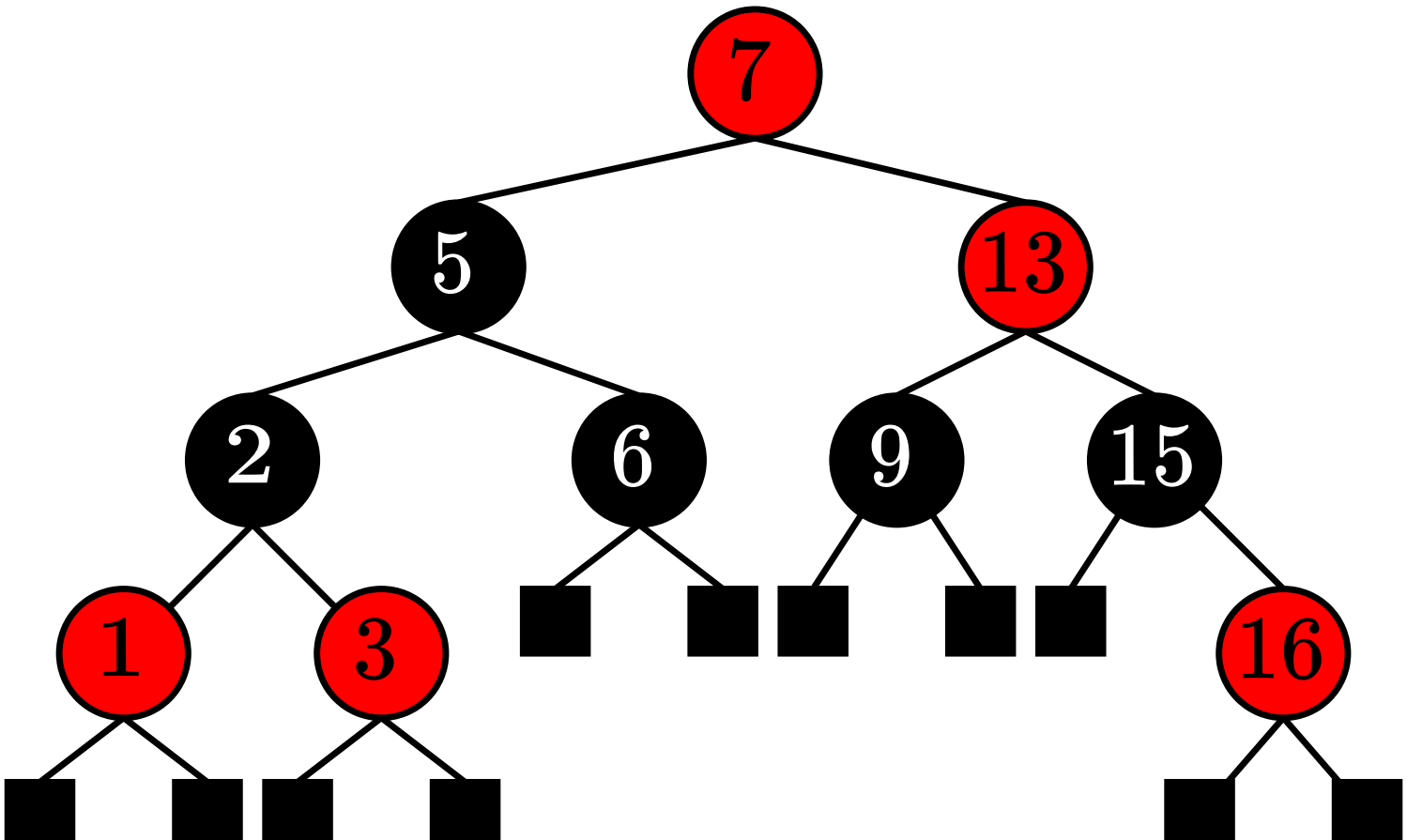
BST – Red-Black Tree (red violation 2)



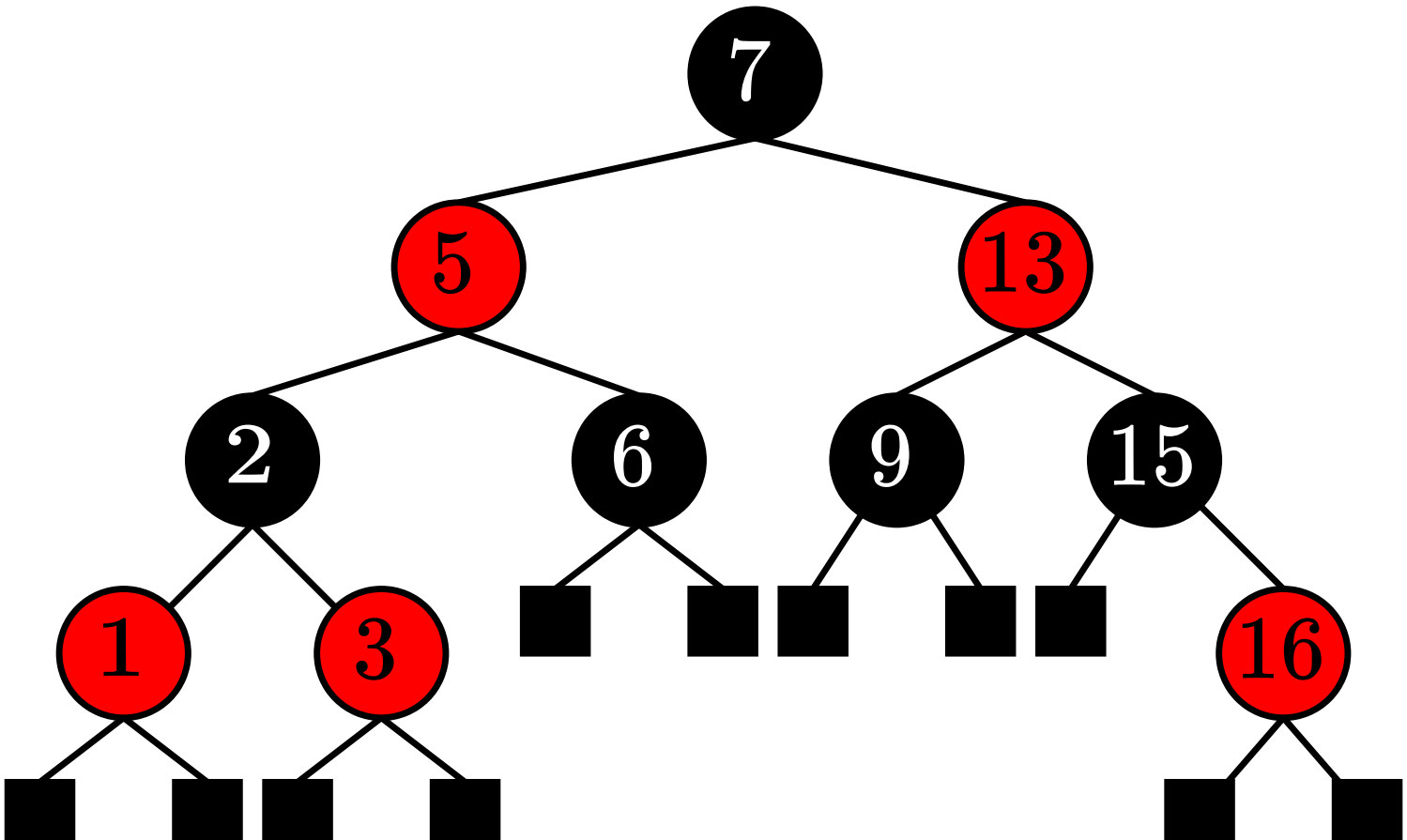
BST – Red-Black Tree (red violation 2)



BST – Red-Black Tree (red&black violation)



BST – Red-Black Tree



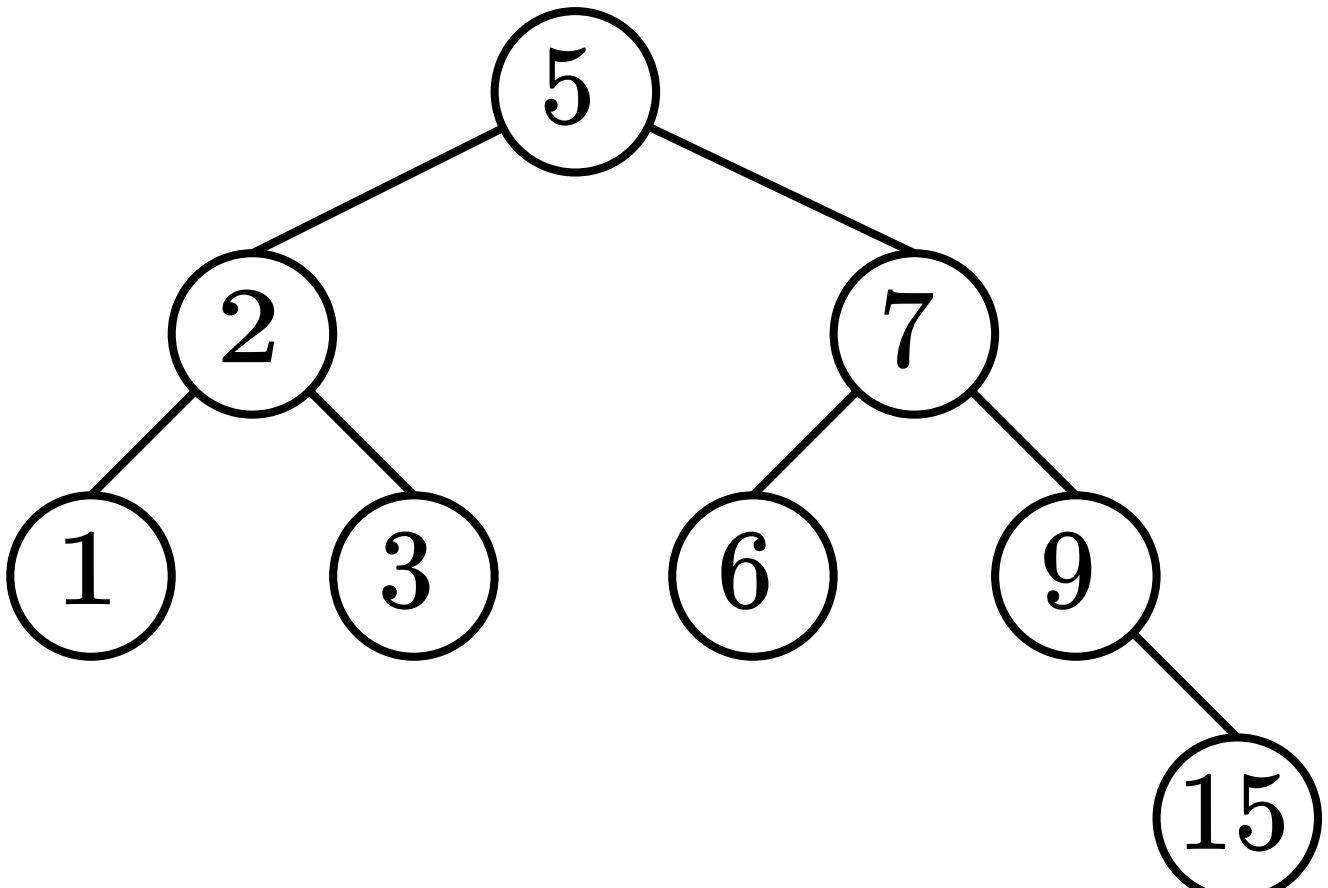
BST – Splay

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

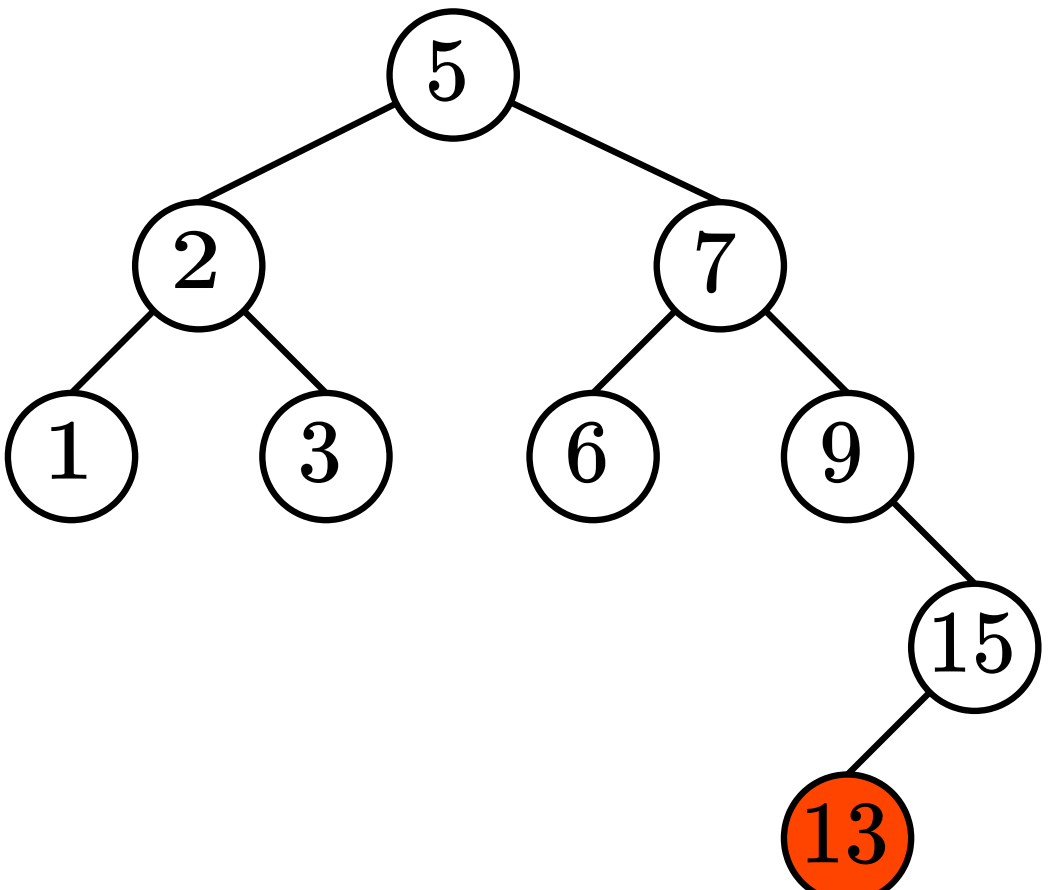
Its runtime is *amortised* $\mathcal{O}(\log n)$ for access and update.

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

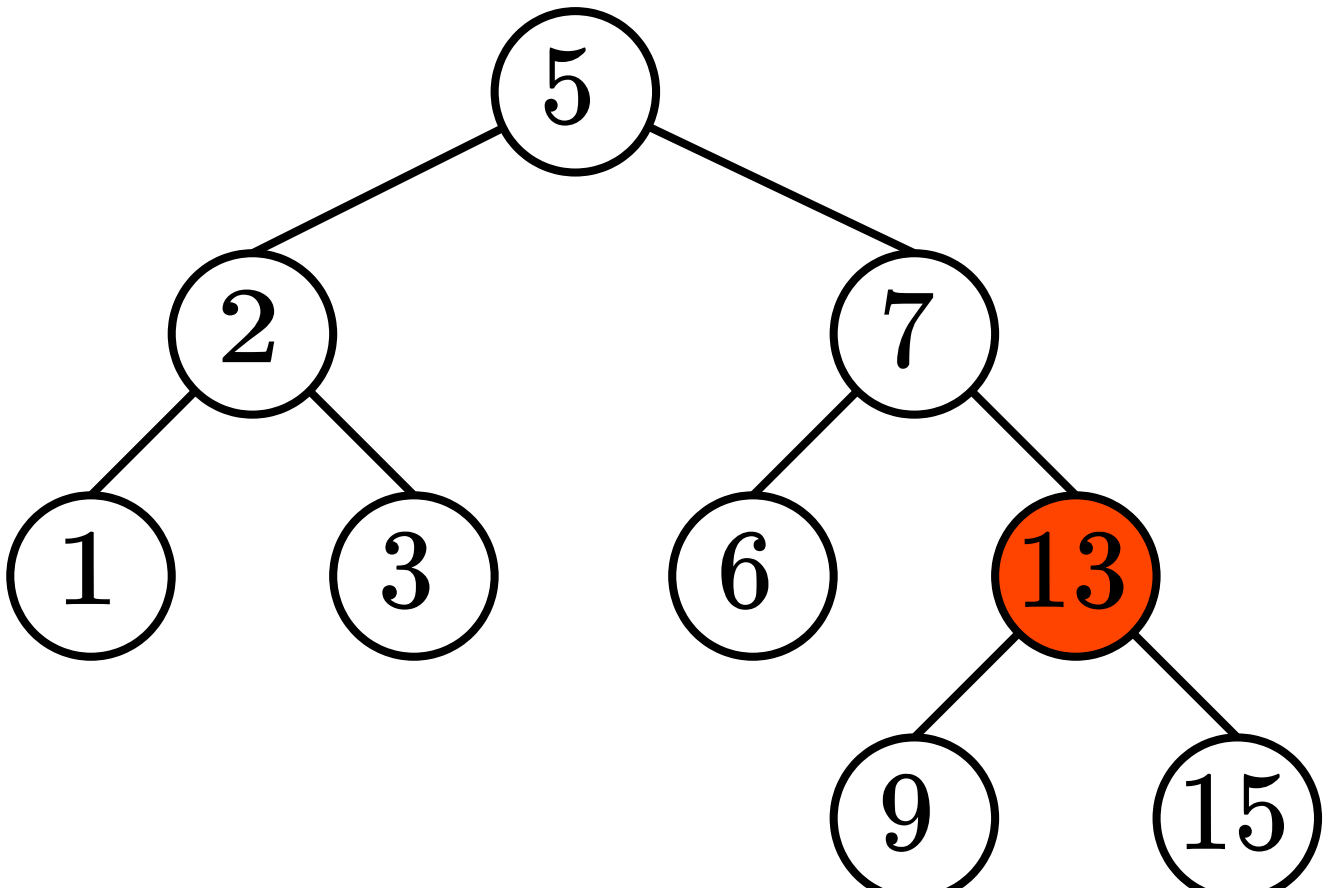
BST – Splay (insertion)



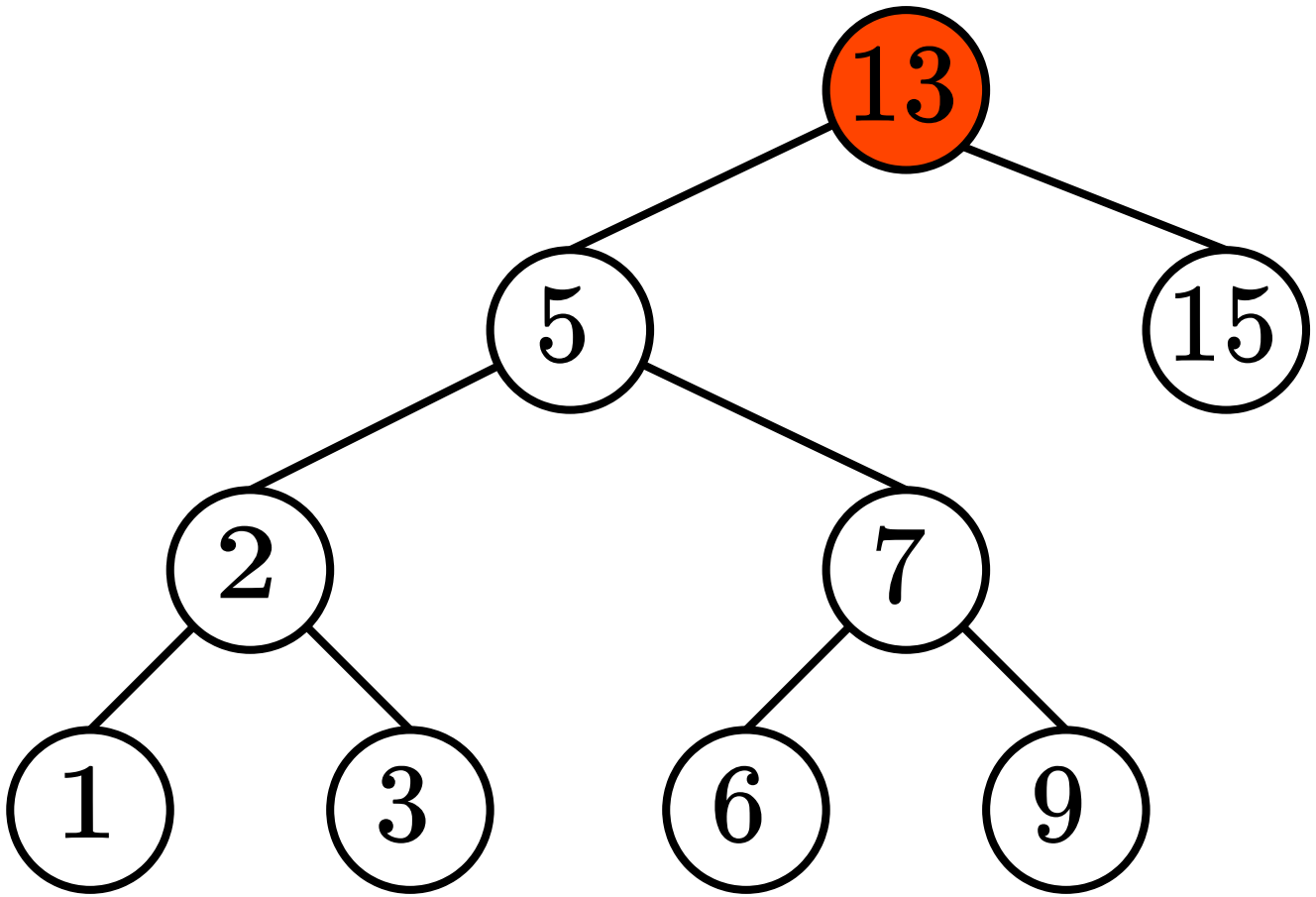
BST – Splay (insertion)



BST – Splay (zag-zig)



BST – Splay (zag-zag)



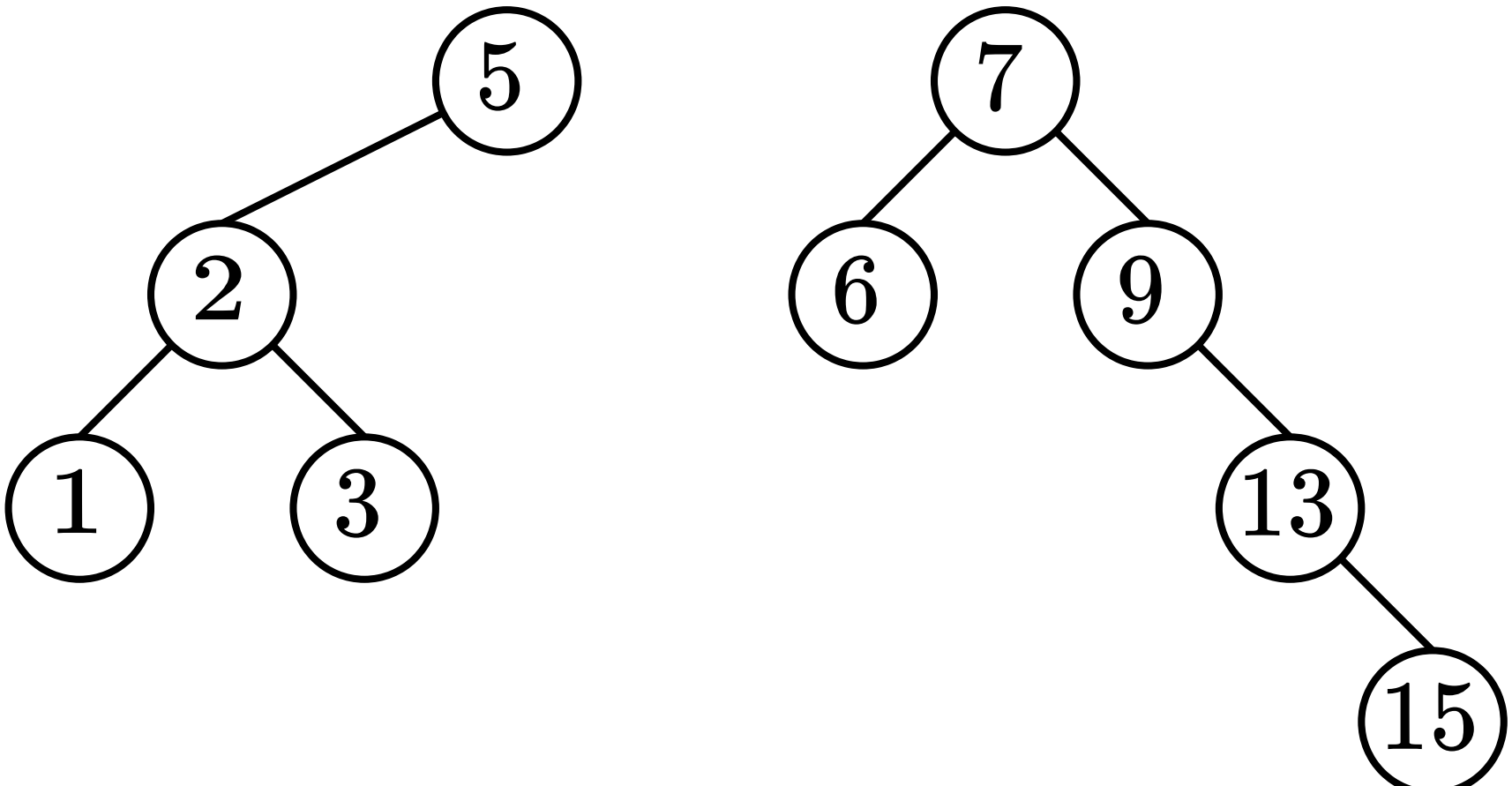
BST – Splay split/join

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

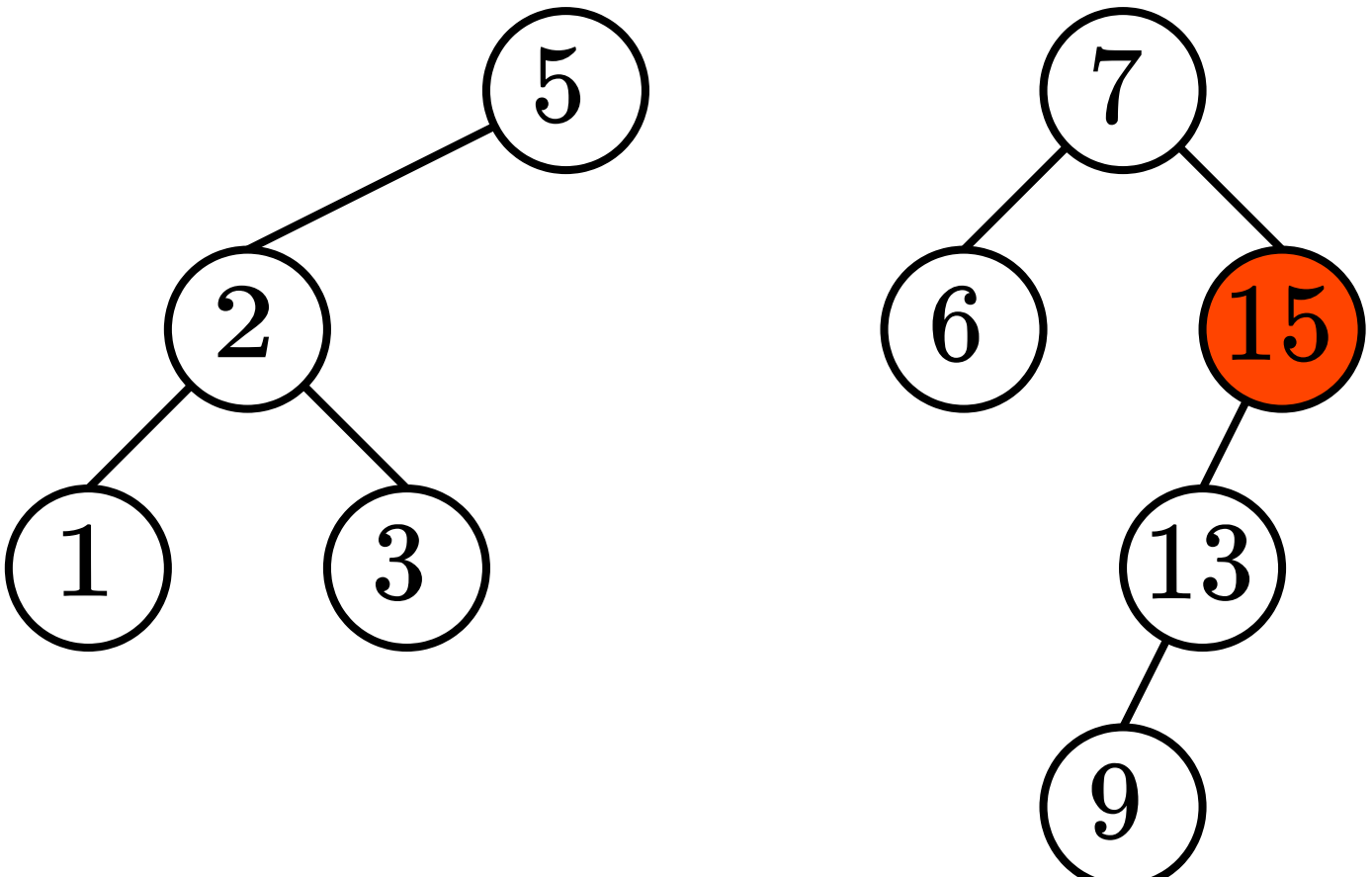
This then allows us to implement all other operations based on join and split.

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

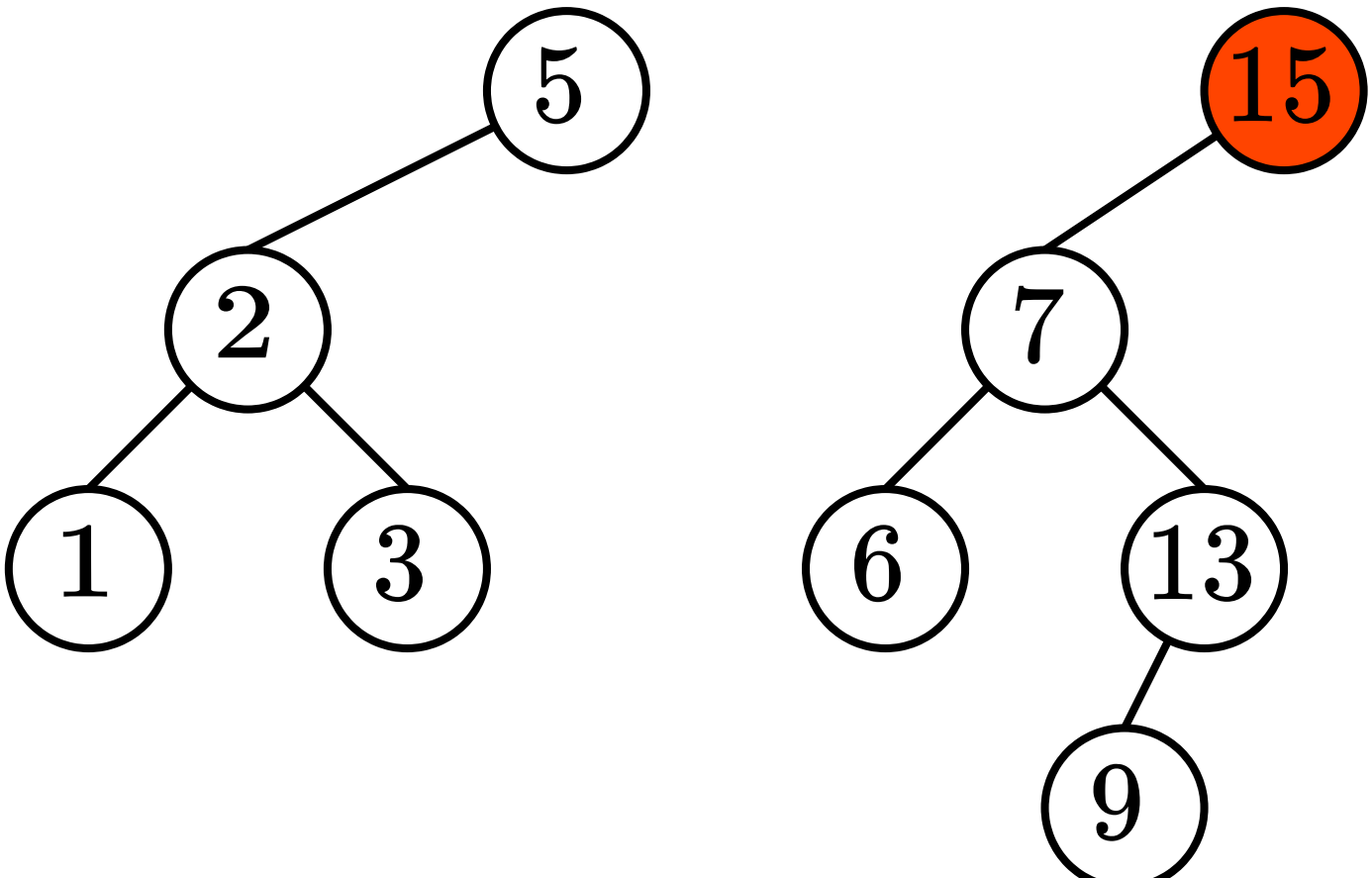
BST – Splay (join)



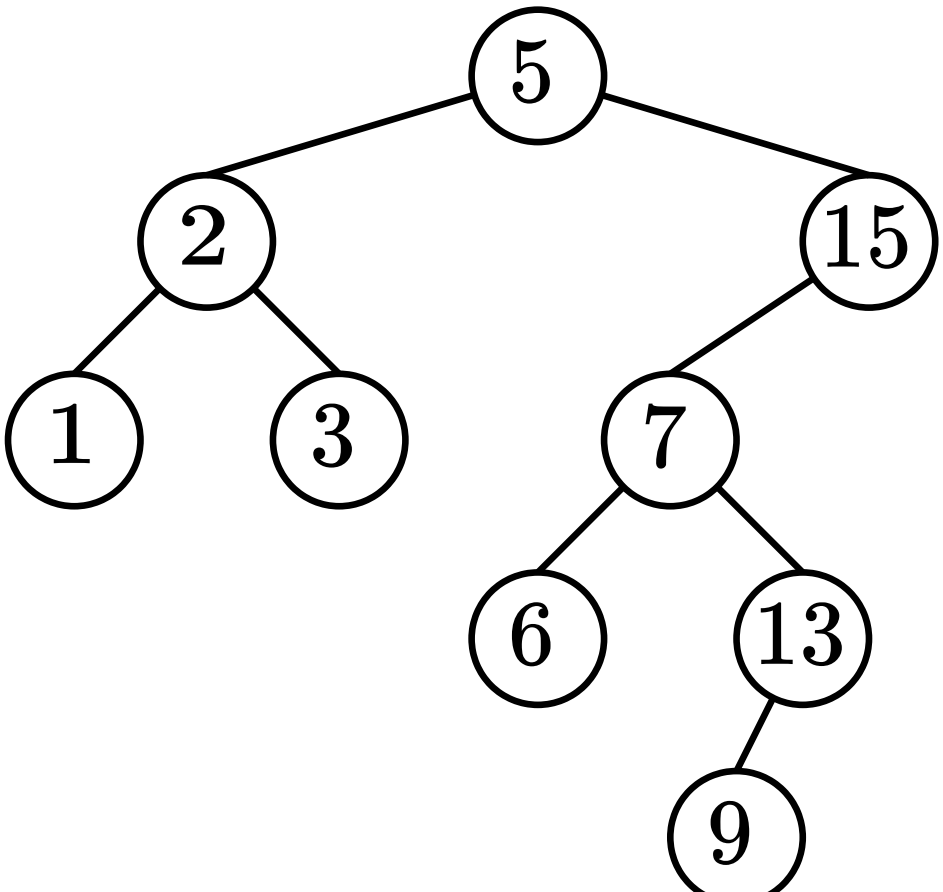
BST – Splay (join)



BST – Splay (join)



BST – Splay (join)



BST – Treap

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

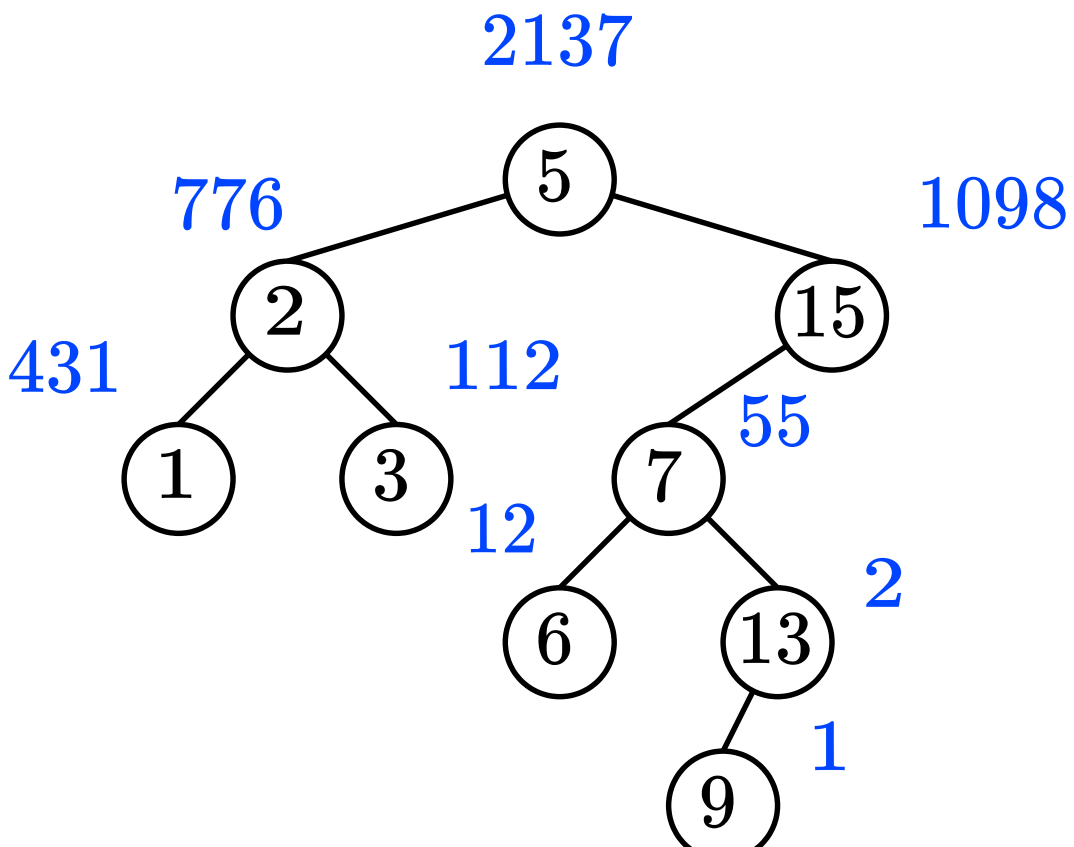
It's based on the following fact:

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

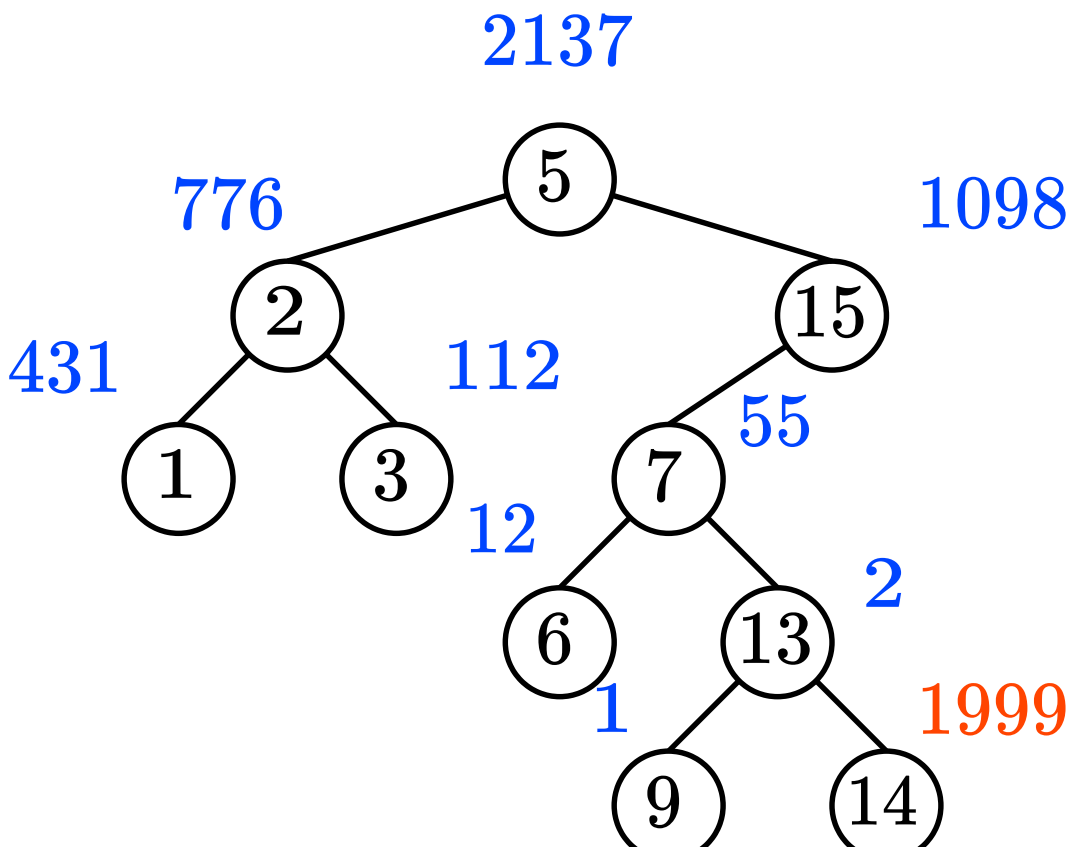
We maintain a BST, but also assign *pseudorandom* weights to each node and maintain the heap invariant (children's weight does not exceed the parent's).

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

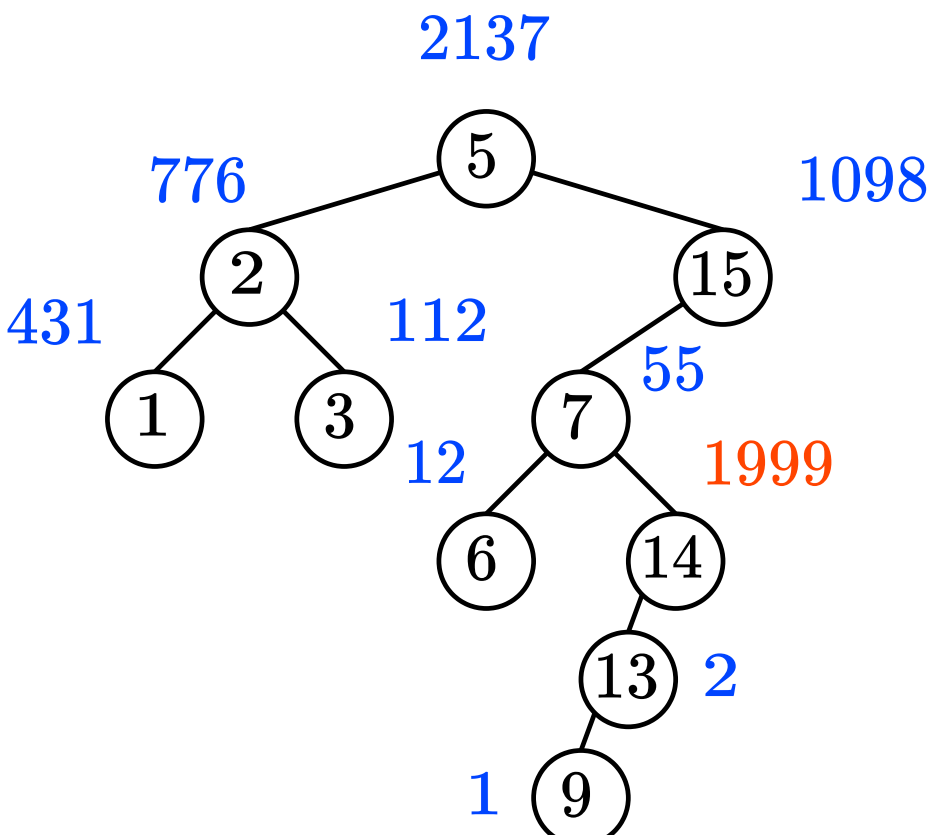
BST – Treap



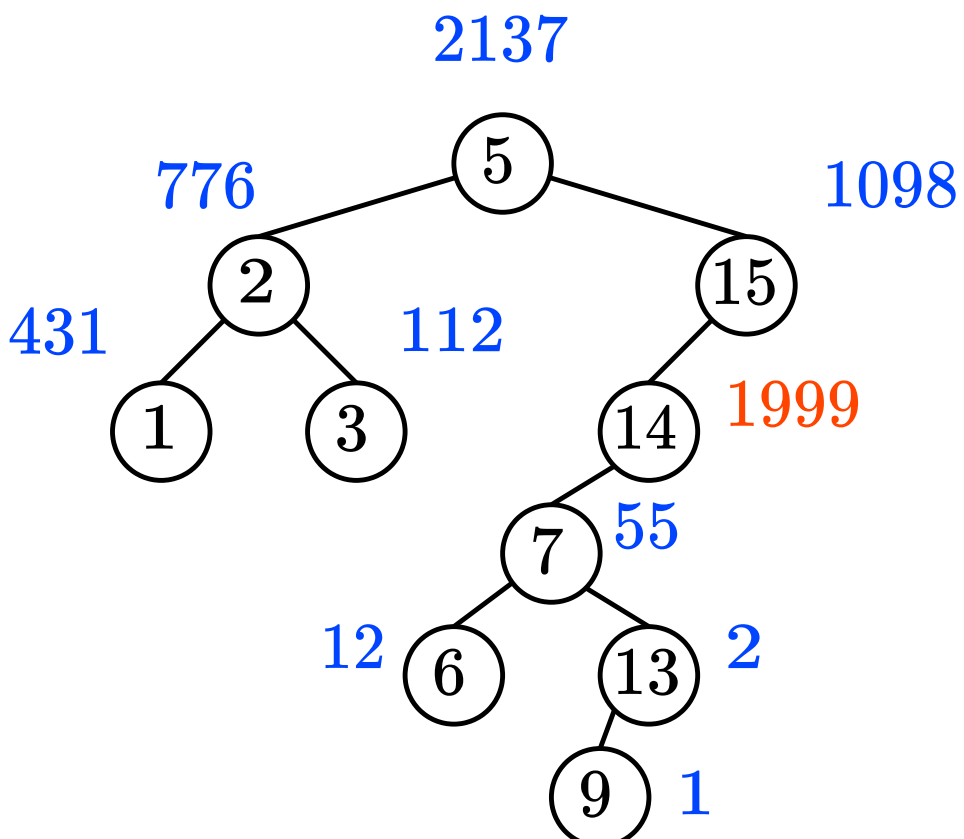
BST – Treap



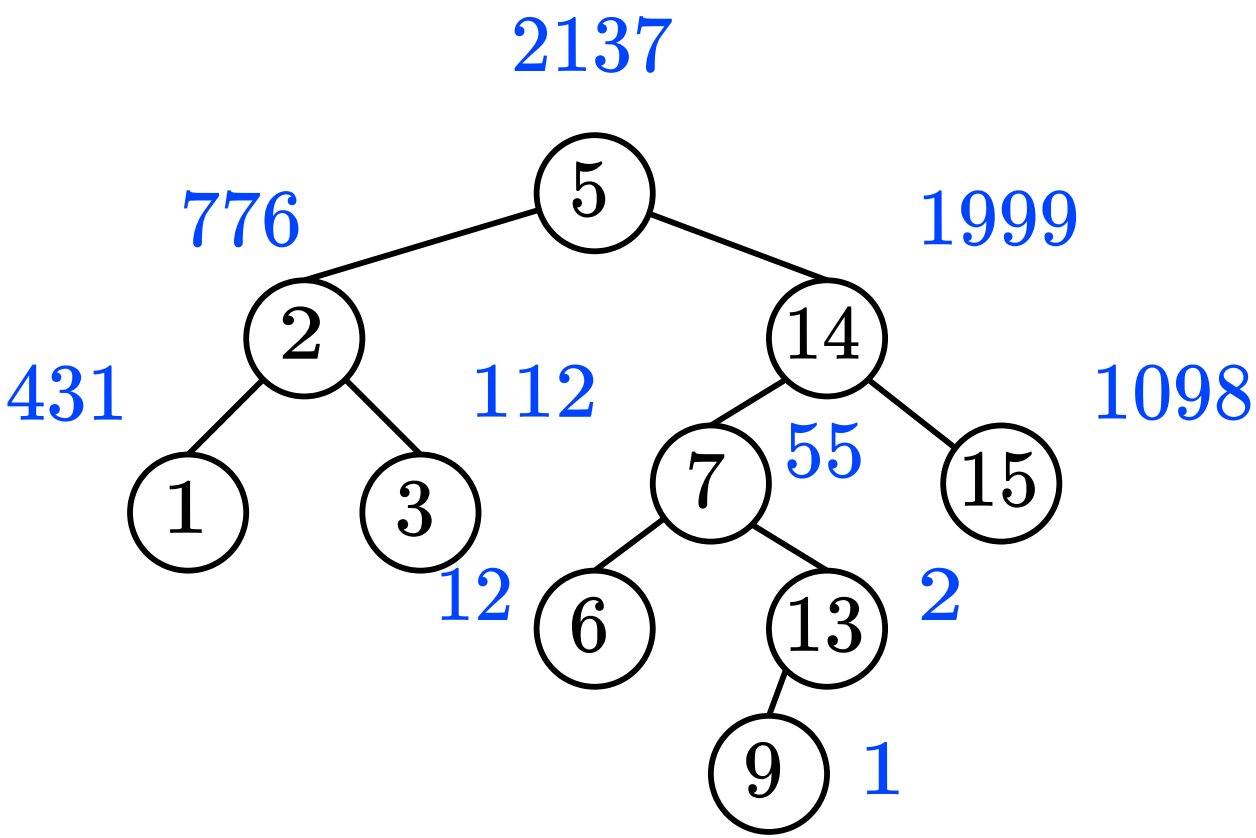
BST – Treap



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:

```
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;
};
```

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 n_from < 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 n_from < 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 n_from < mid
    agg ⊕= query_at(q_from, min(q_to, mid), 2 * node, n_from, mid)
  else
    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)

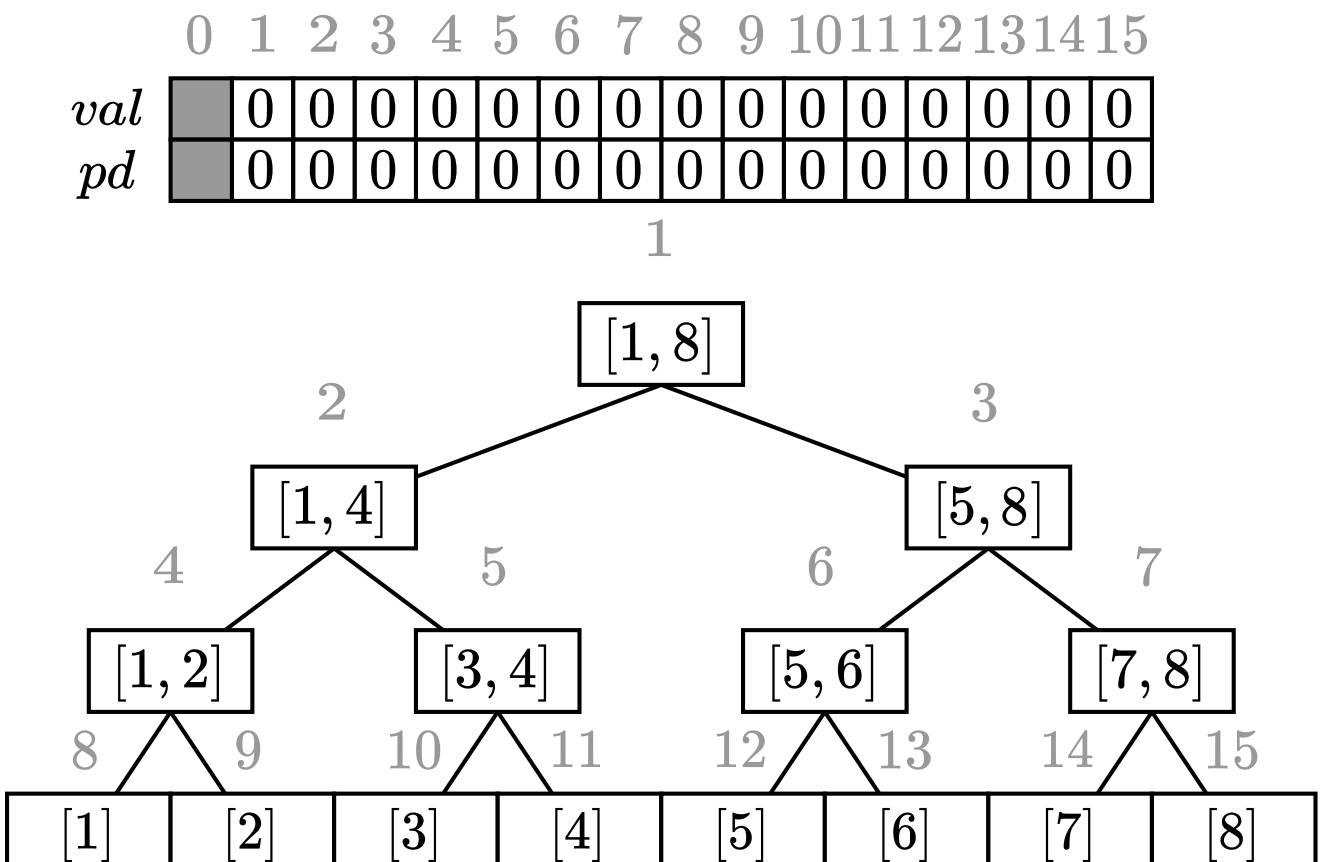
```
fn update_at(q_from, q_to, node, n_from, n_to, value)
  if q_from == n_from and q_to == n_to
    pushdown[node] ⊕= value
  mid = midpoint(n_from, n_to)
  if n_from < mid
    update_at(q_from, min(q_to, mid), 2 * node, n_from, mid, value)
  else
    update_at(max(q_to, mid), q_to, 2 * node + 1, mid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

BST – Pushdown values (range query)

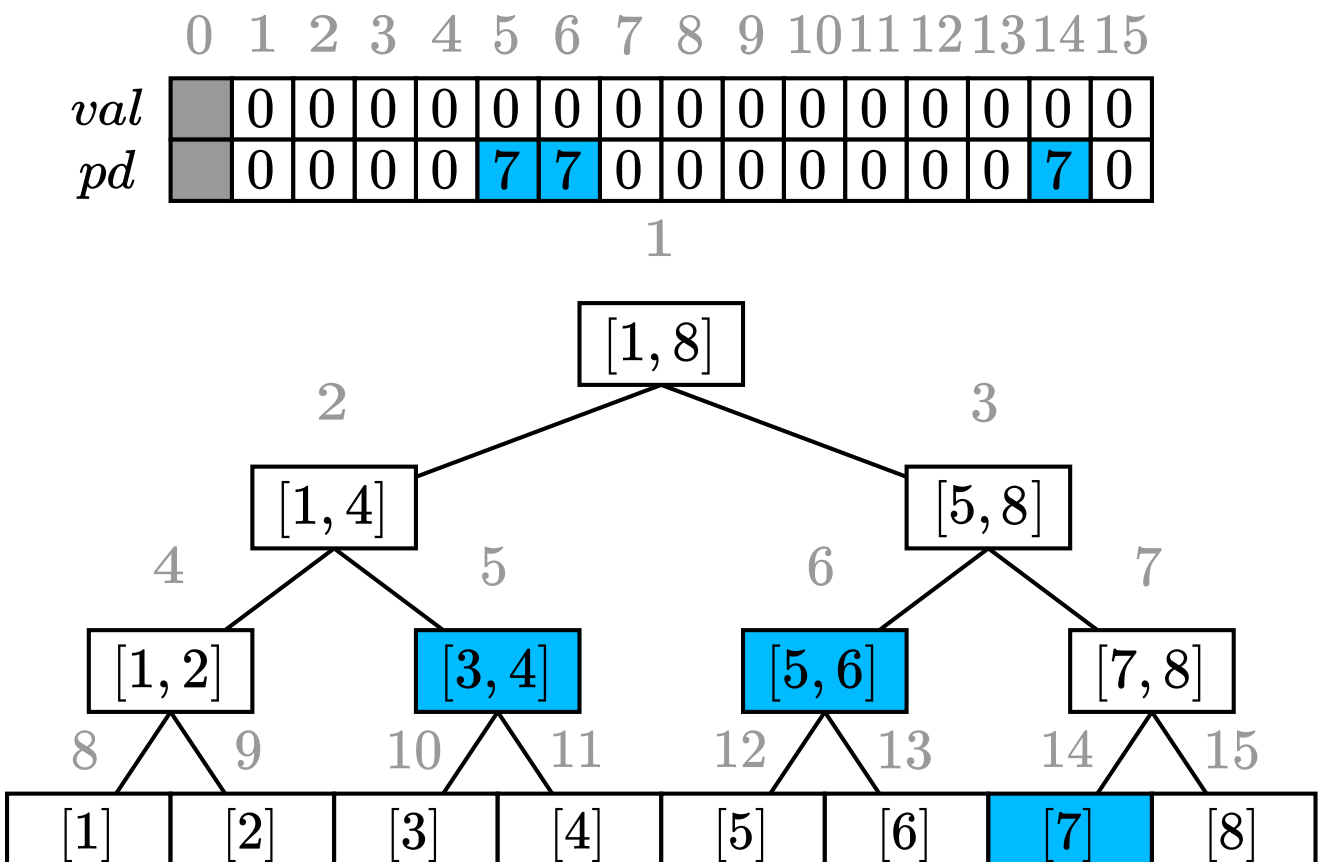
```
fn push_down(node, n_from, n_to)
  tree[node] ⊕= pushdown[node] * (n_to - n_from + 1)
  if n_from != n_to
    pushdown[2 * node] ⊕= pushdown[node]
    pushdown[2 * node + 1] ⊕= pushdown[node]
  pushdown[node] = 0

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
```

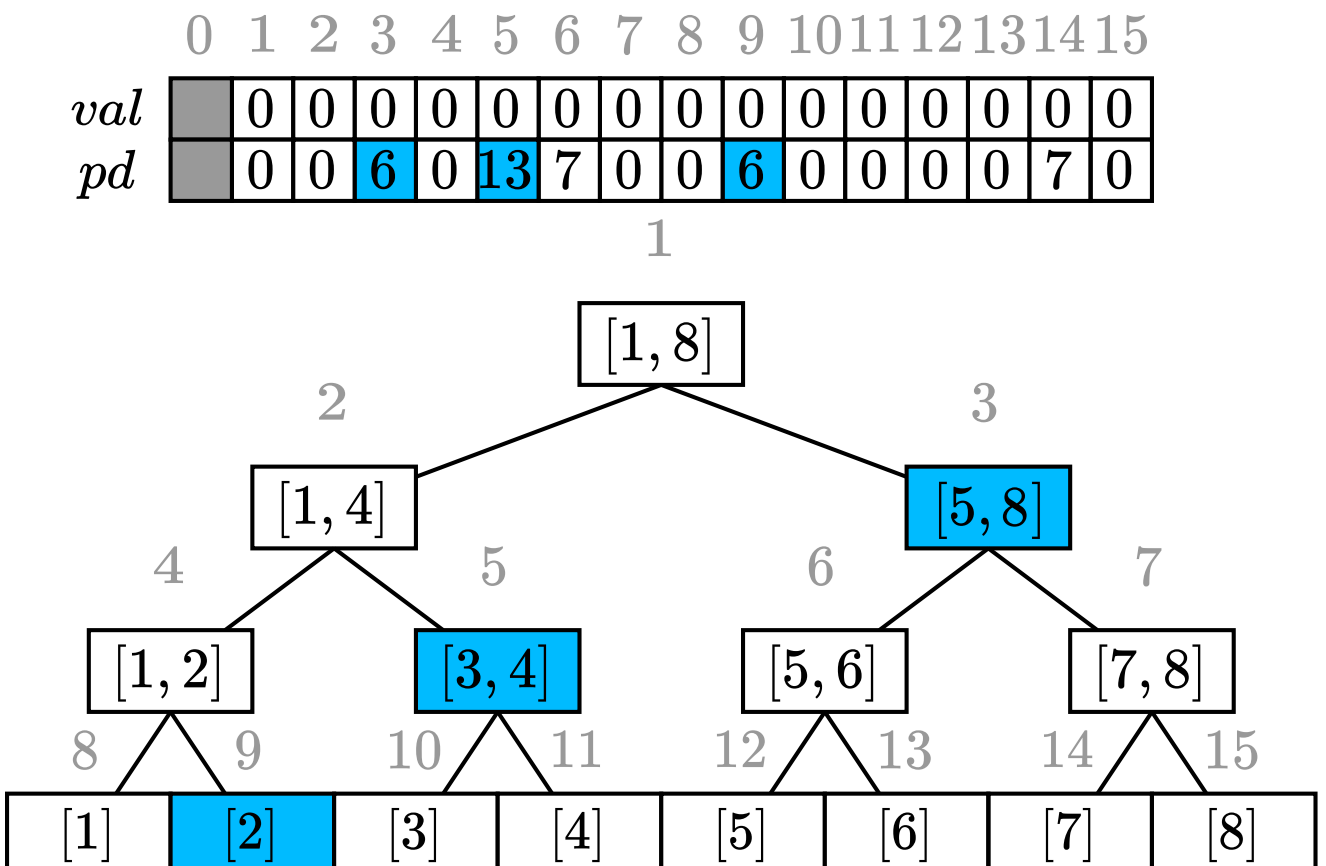
BST – Segment trees



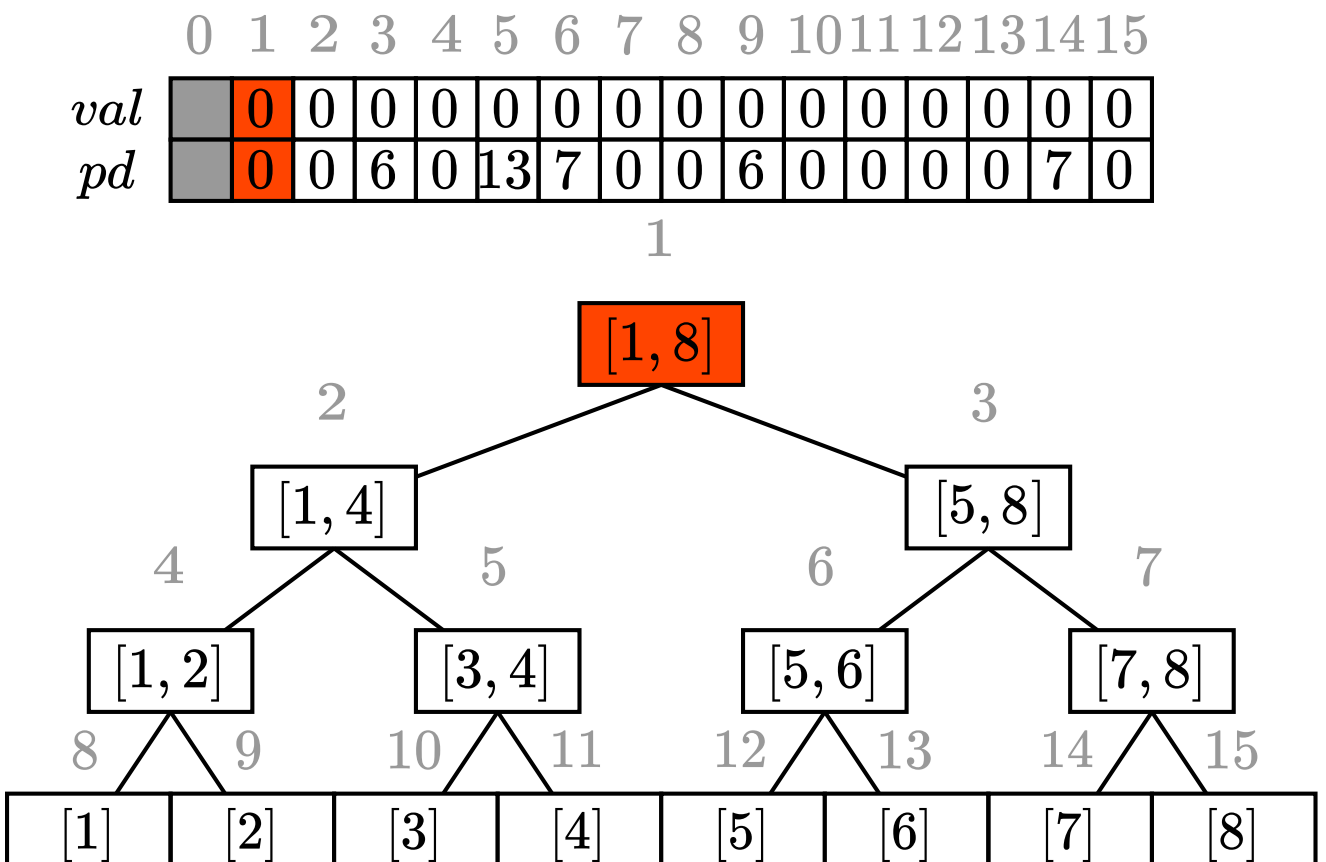
BST – Segment trees (update $[3, 7] + 7$)



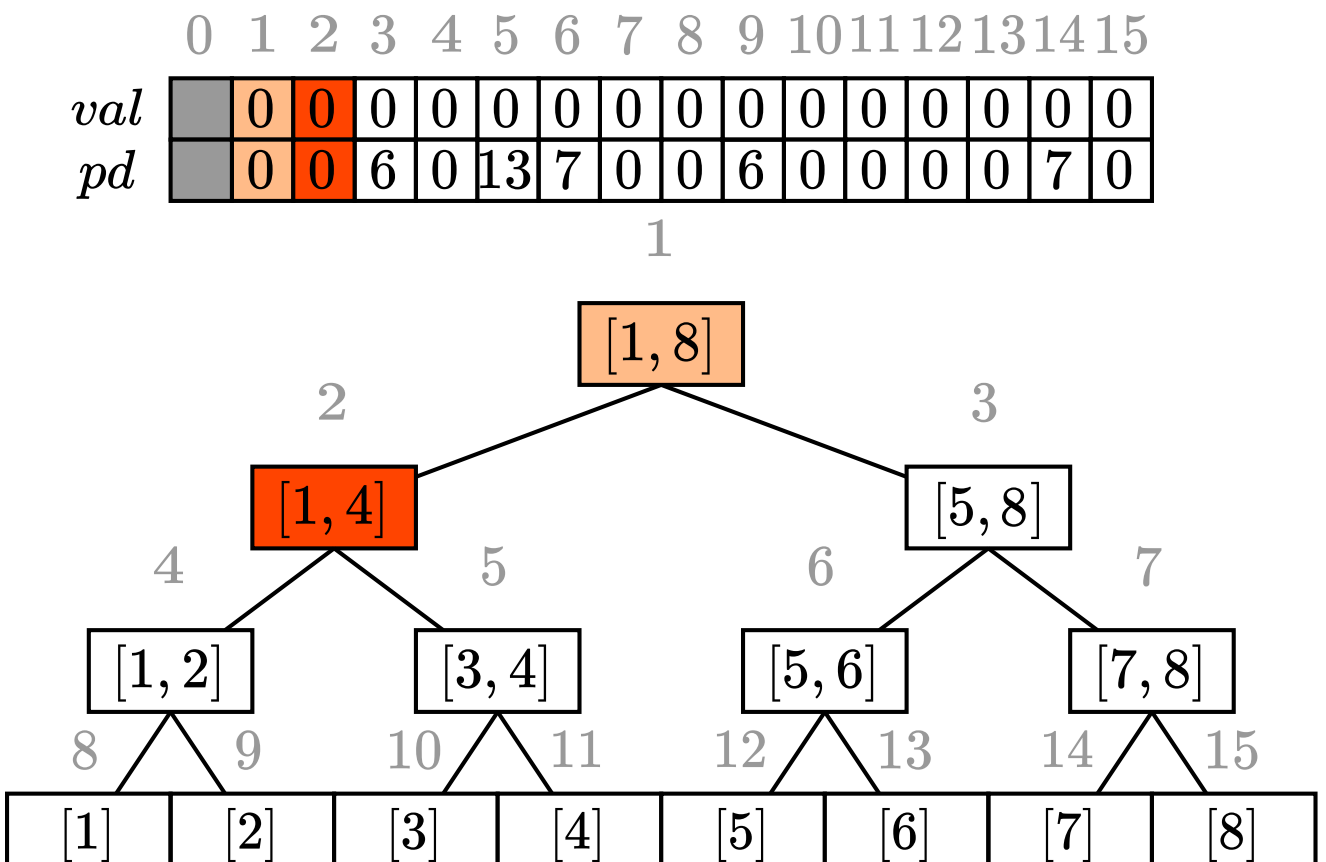
BST – Segment trees (update $[2, 16] + 6$)



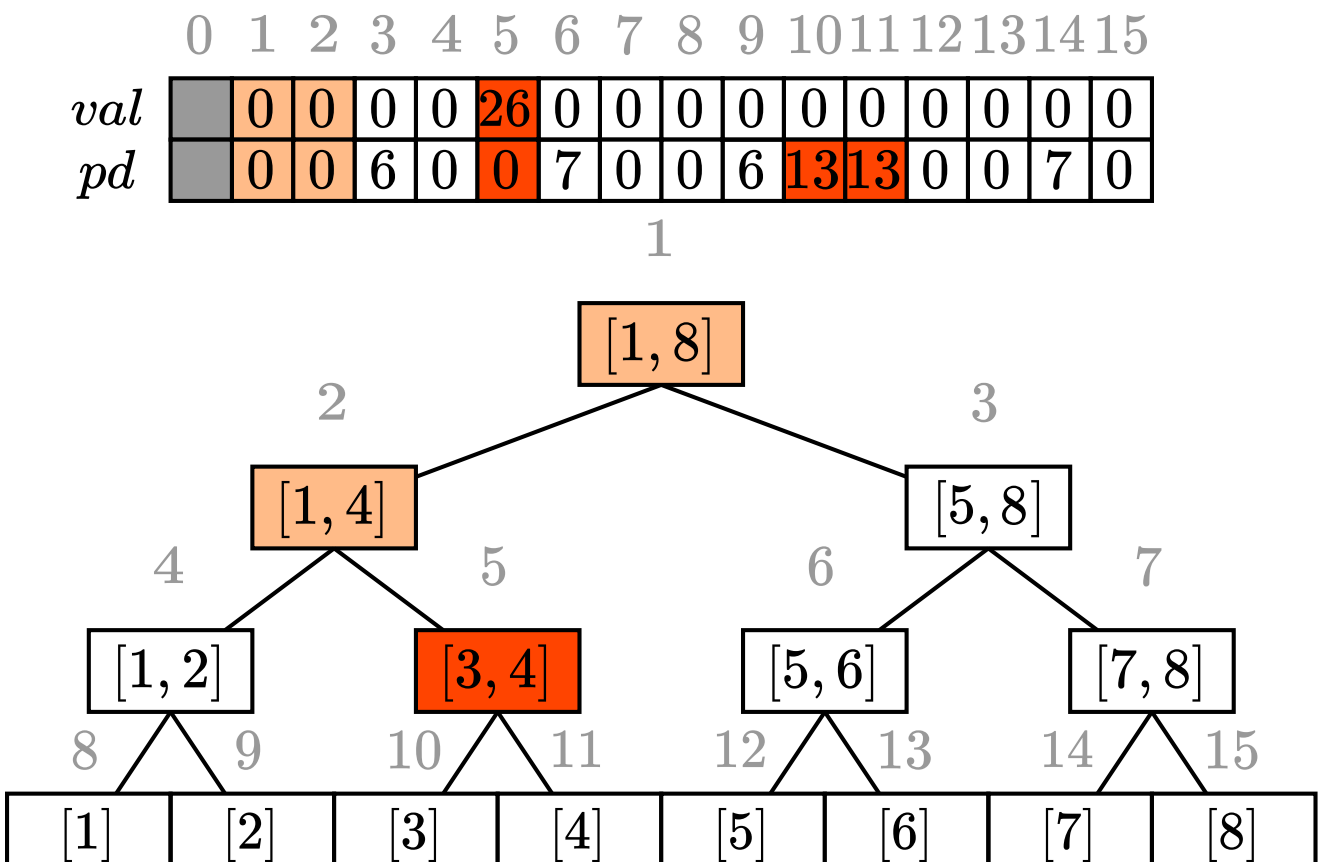
BST – Segment trees (query $\sum[4, 6]$)



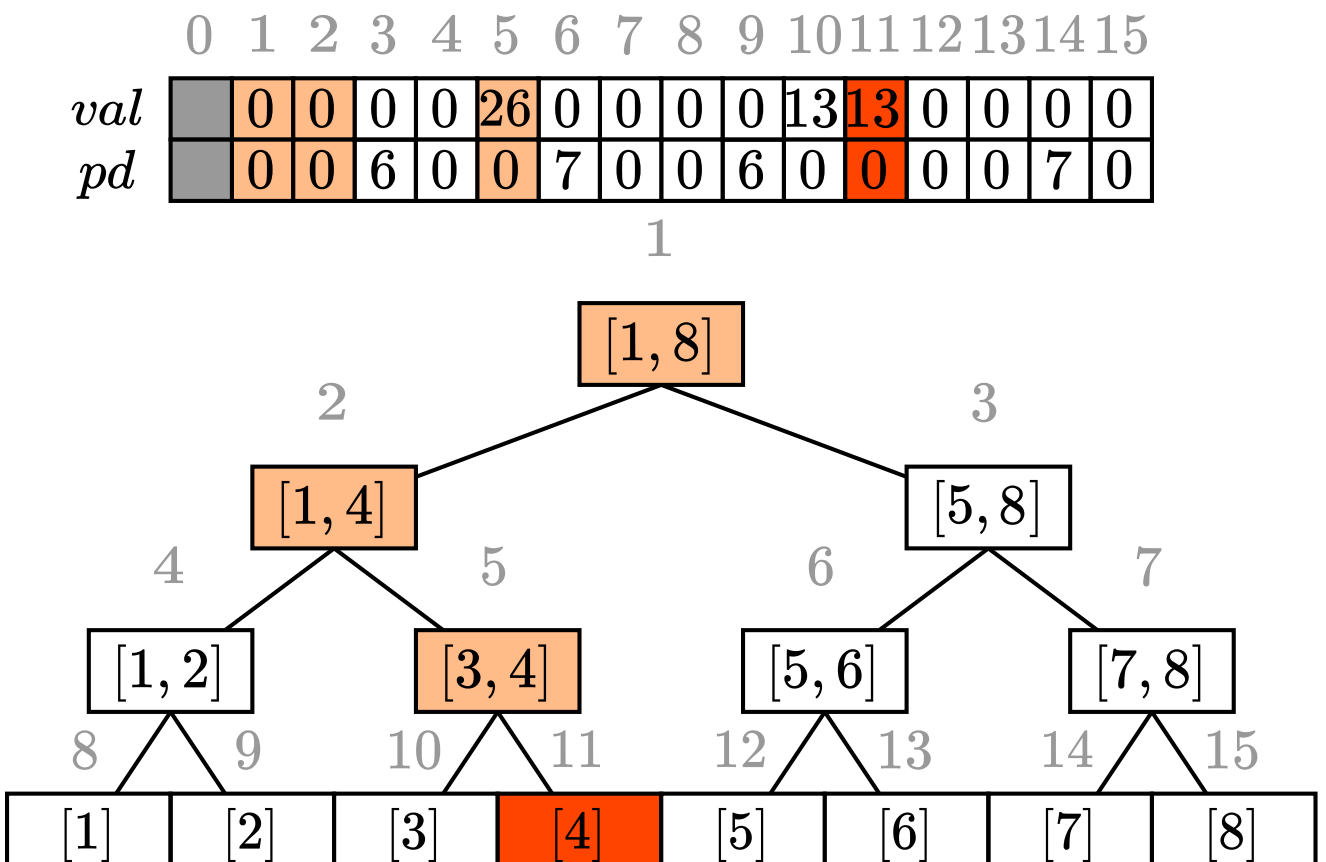
BST – Segment trees (query $\sum[4, 6]$)



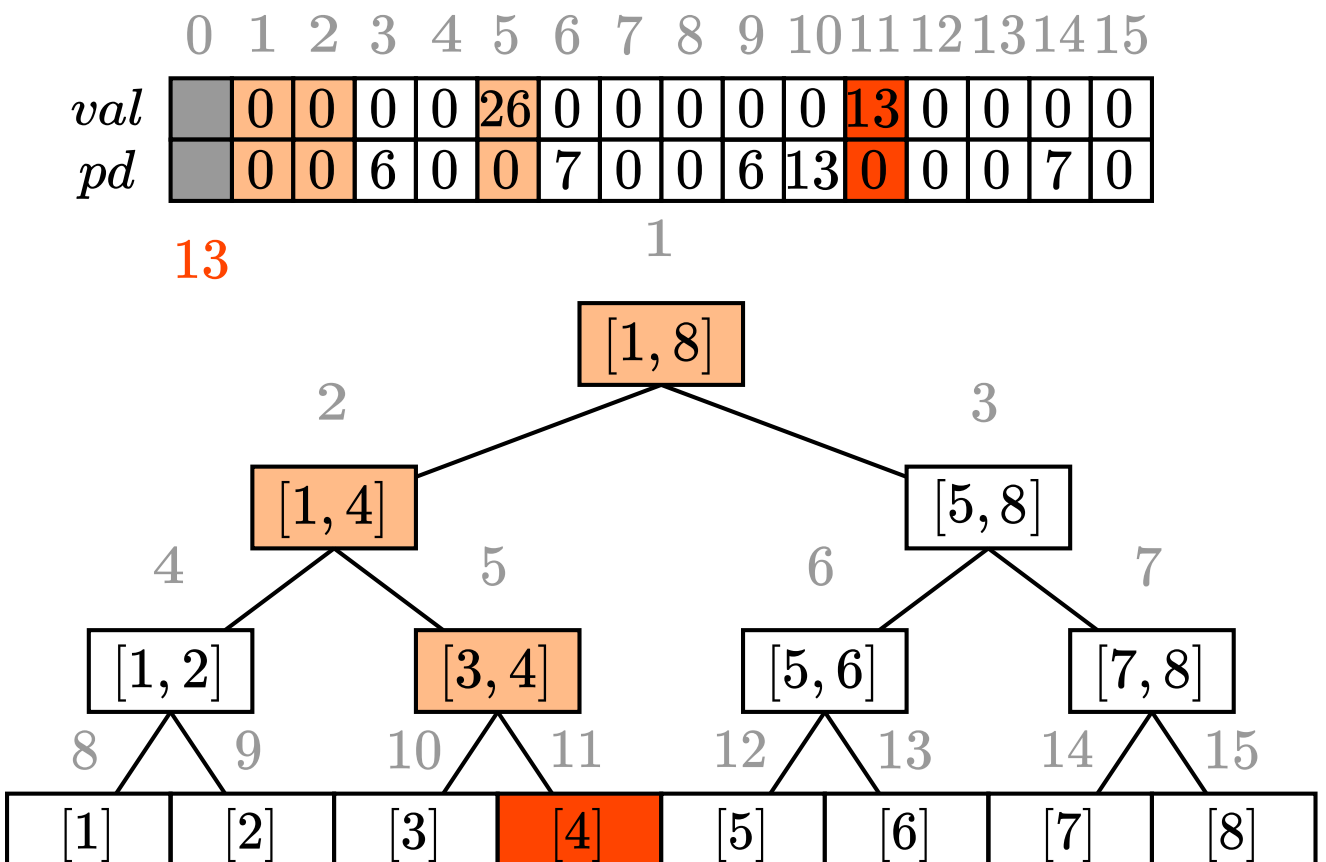
BST – Segment trees (query $\sum[4, 6]$)



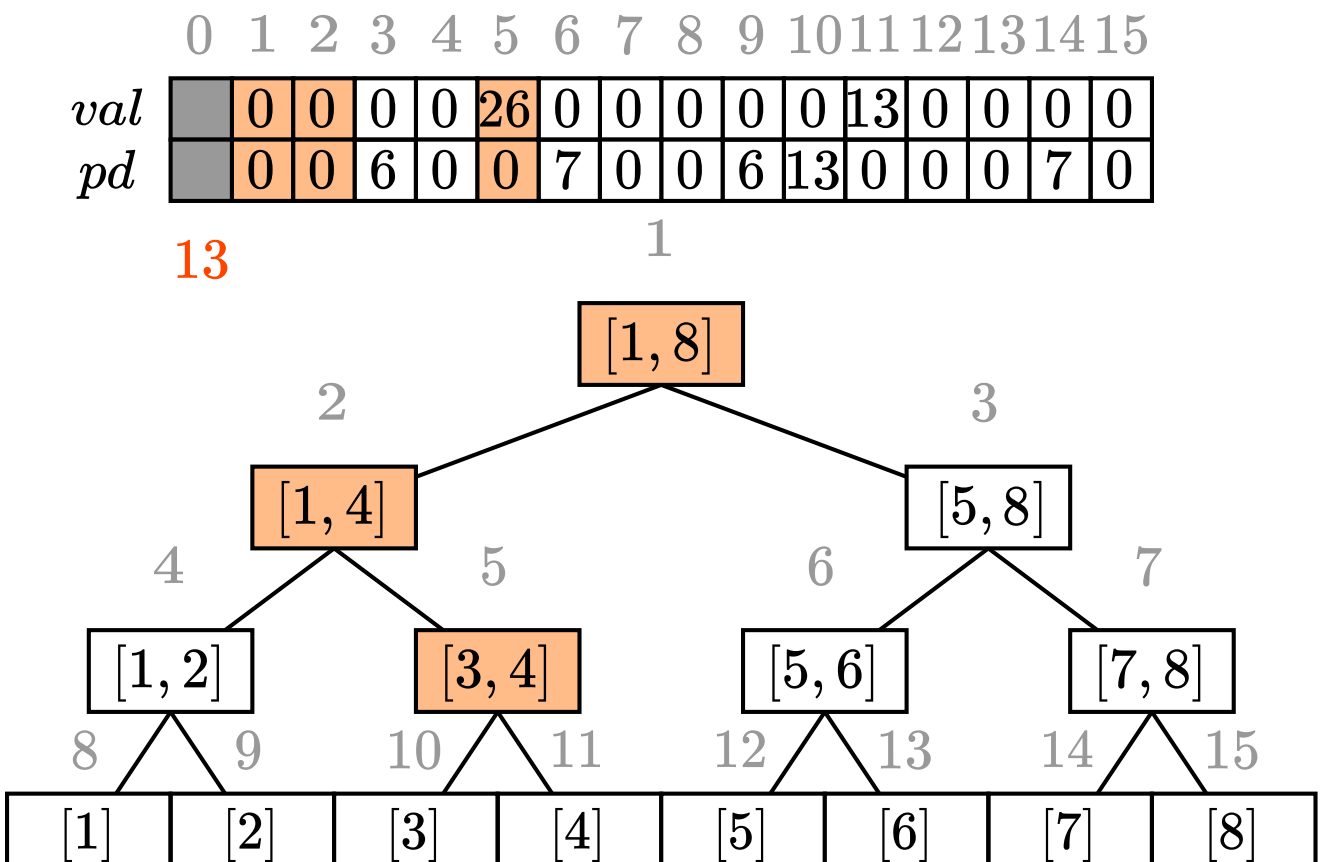
BST – Segment trees (query $\sum[4, 6]$)



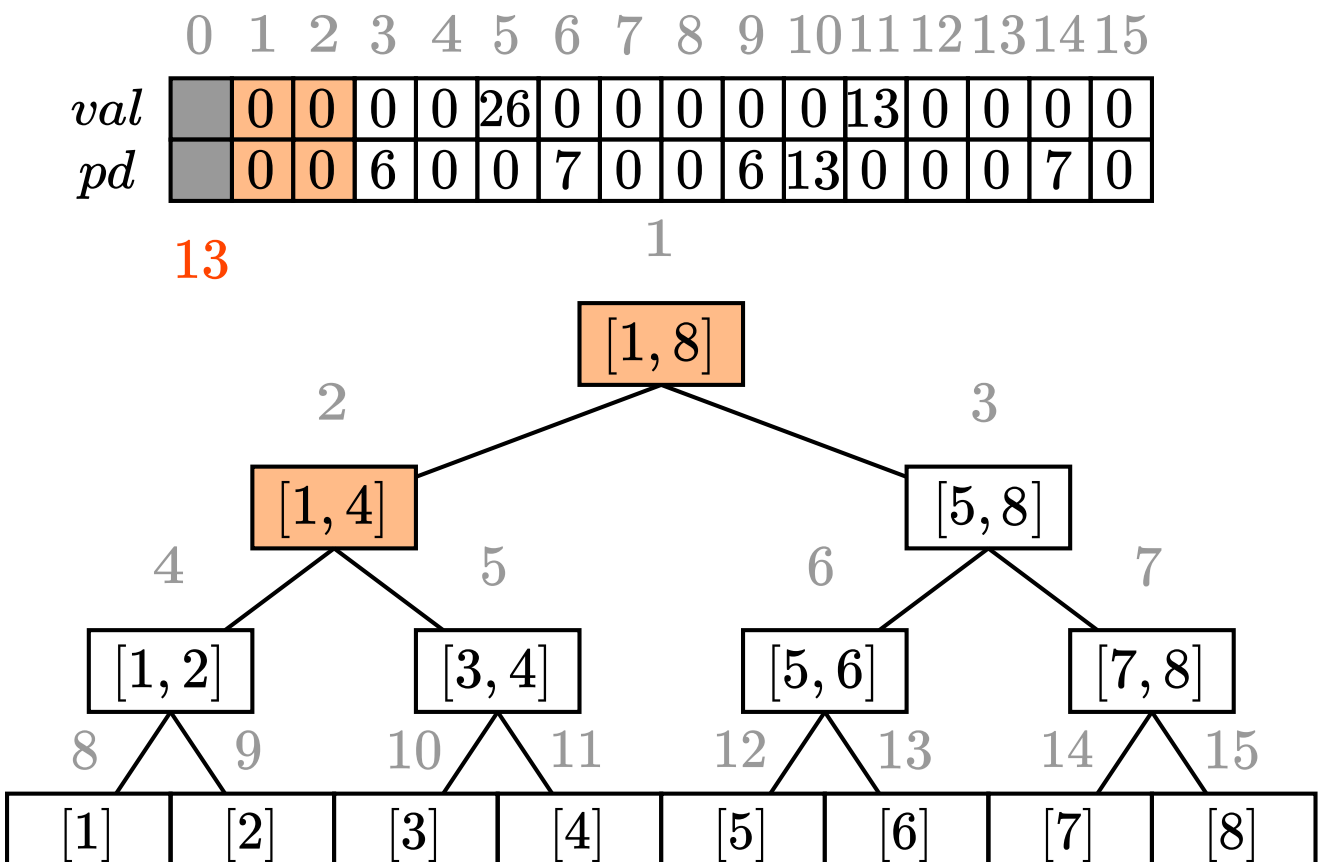
BST – Segment trees (query $\sum[4, 6]$)



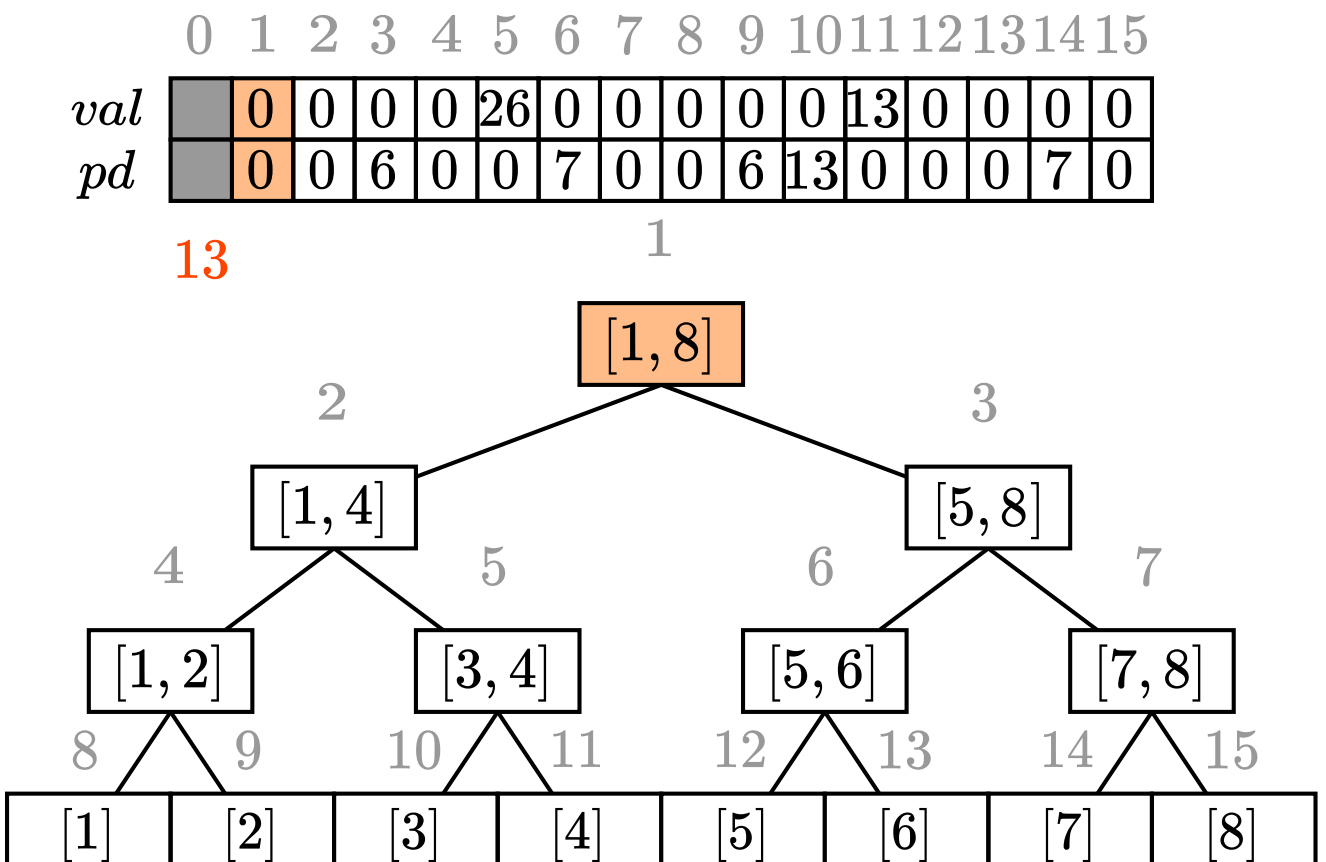
BST – Segment trees (query $\sum[4, 6]$)



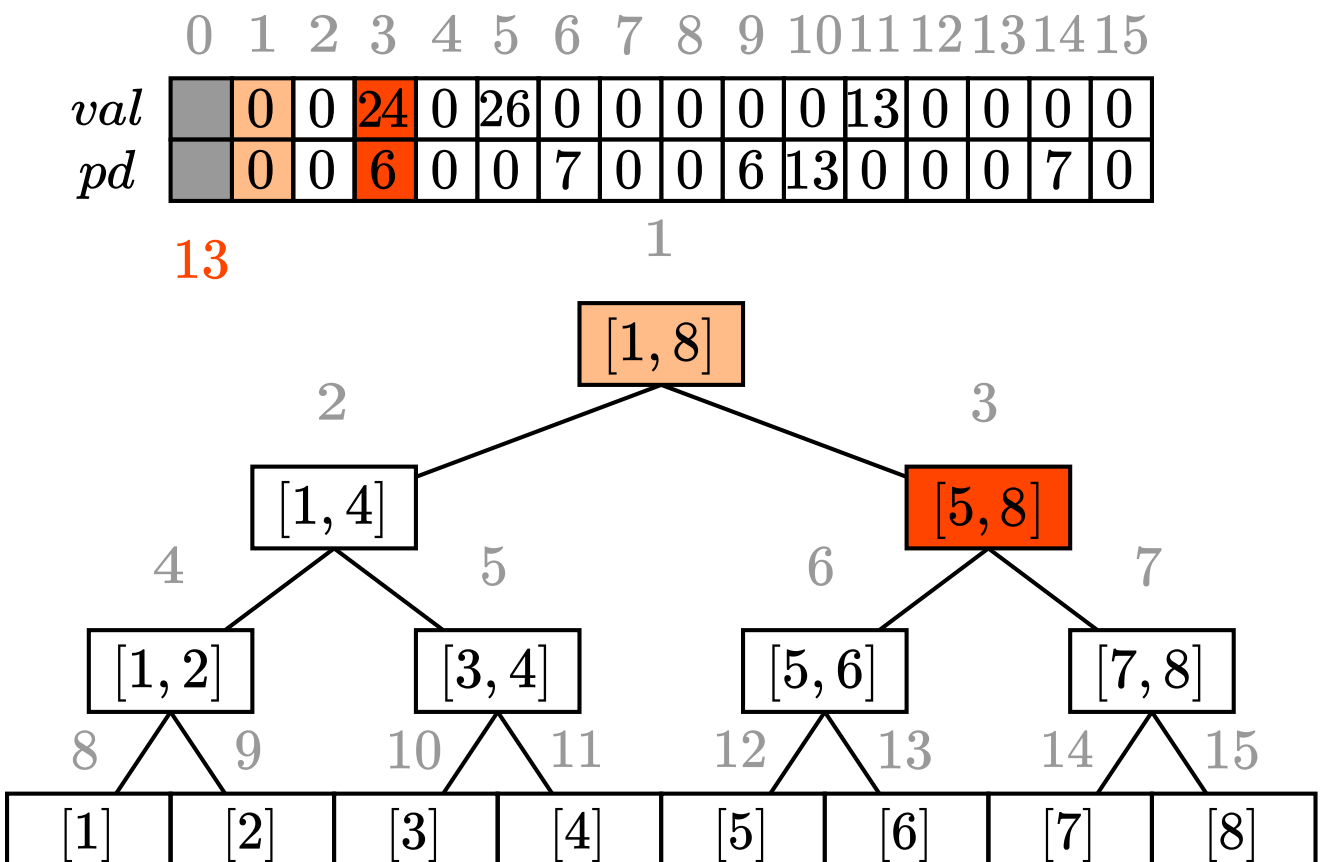
BST – Segment trees (query $\sum[4, 6]$)



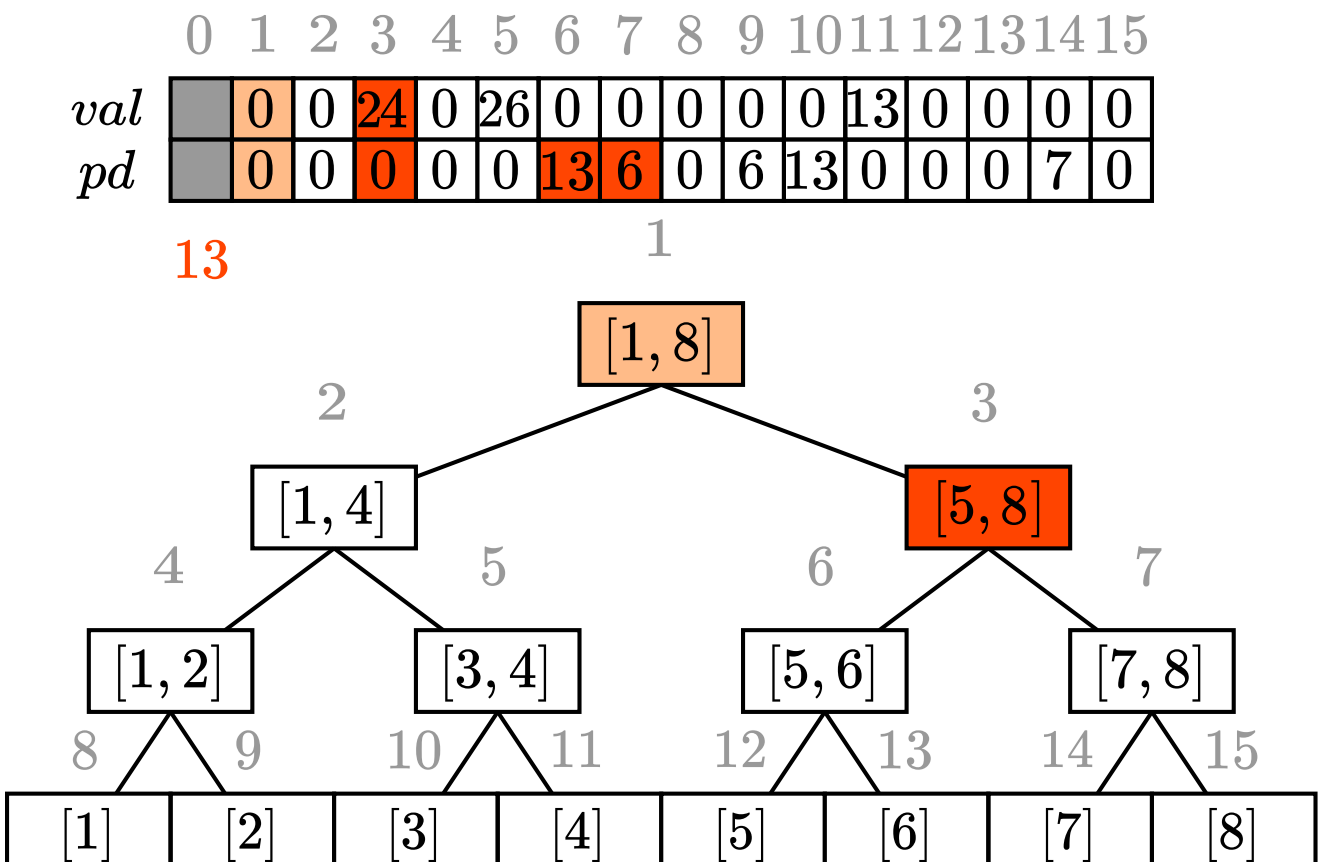
BST – Segment trees (query $\sum[4, 6]$)



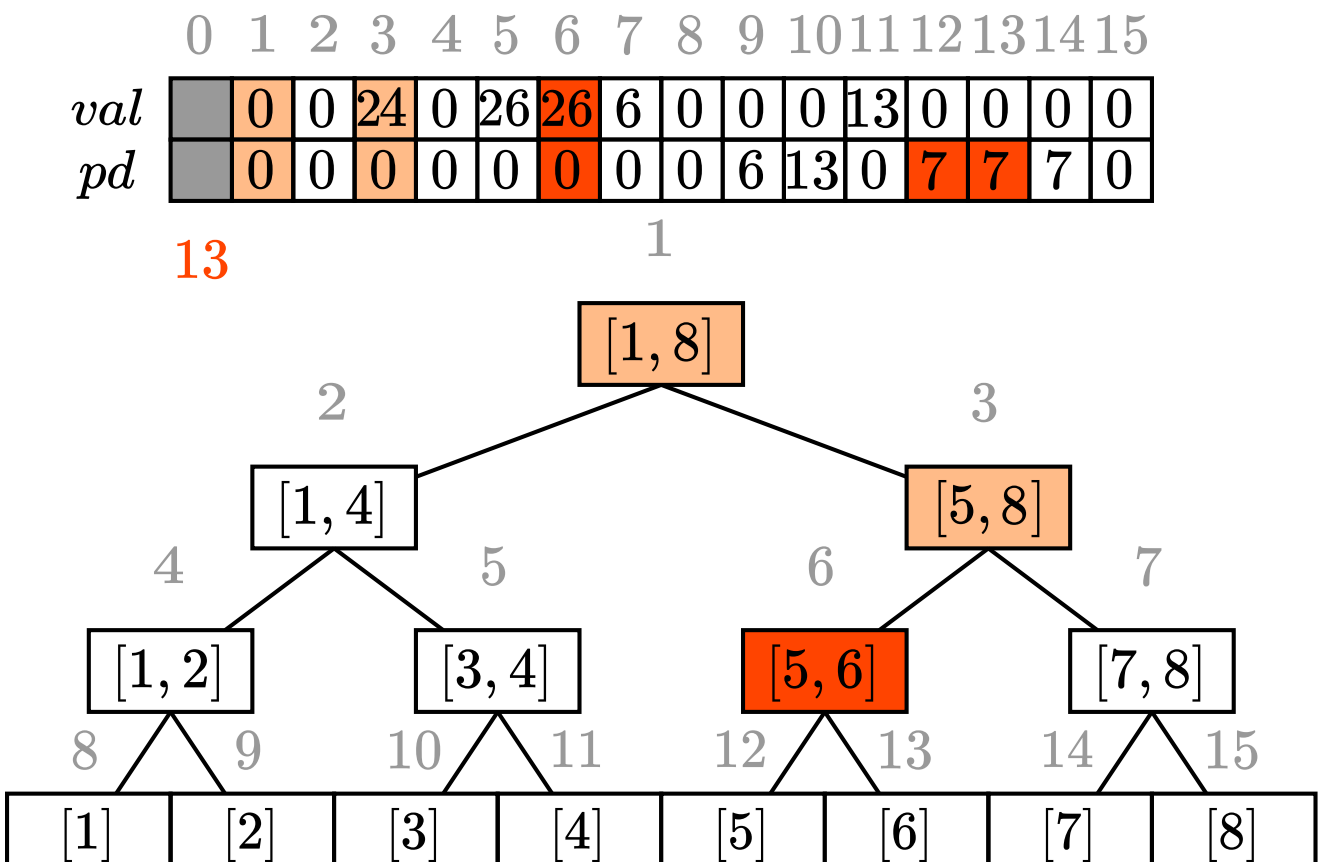
BST – Segment trees (query $\sum[4, 6]$)



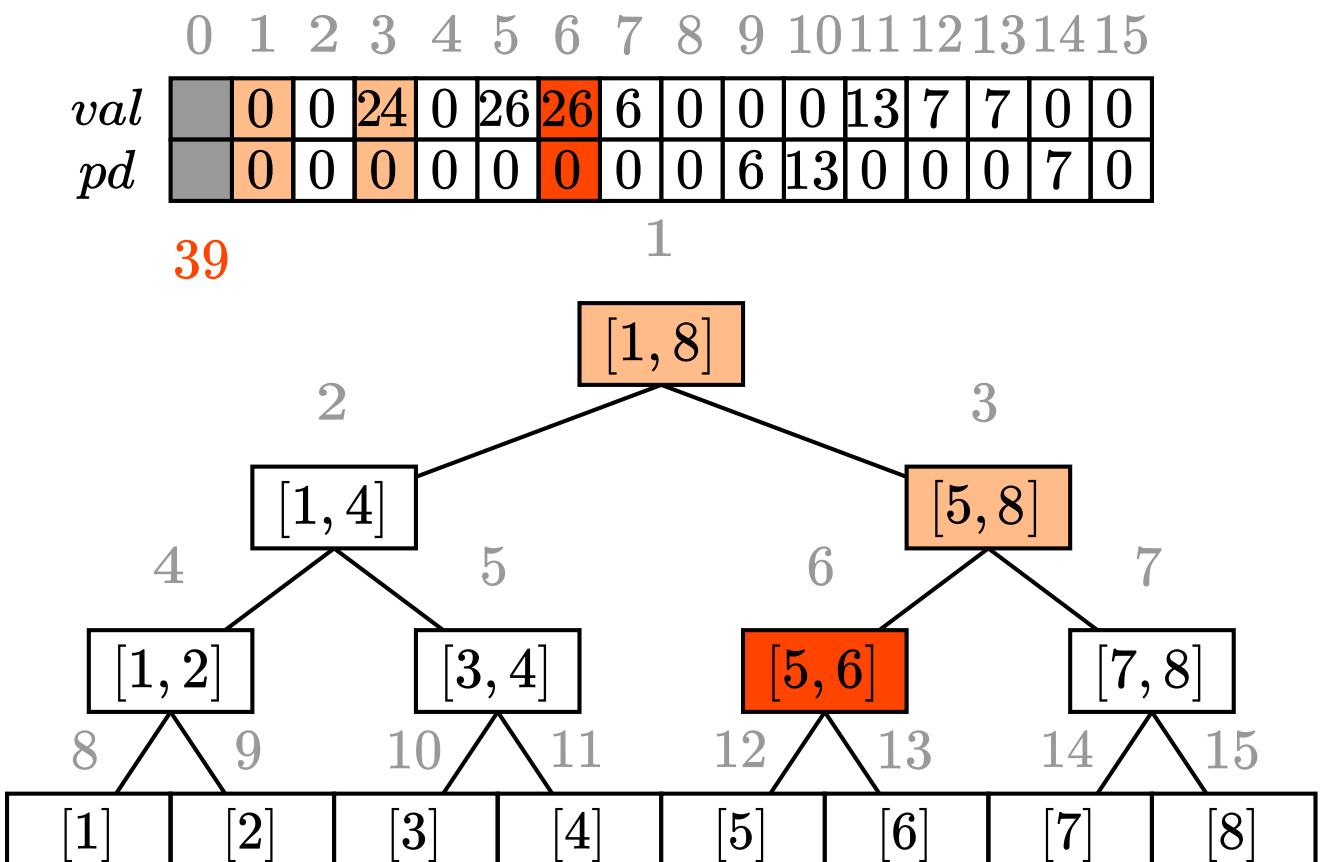
BST – Segment trees (query $\sum[4, 6]$)



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



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



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 $\mathcal{O}\left(\log^k n\right)$. 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

PSS and BUR: 01.07.2025, 10:00 AM

Good luck!