COMP 2402

Random Binary Search Trees

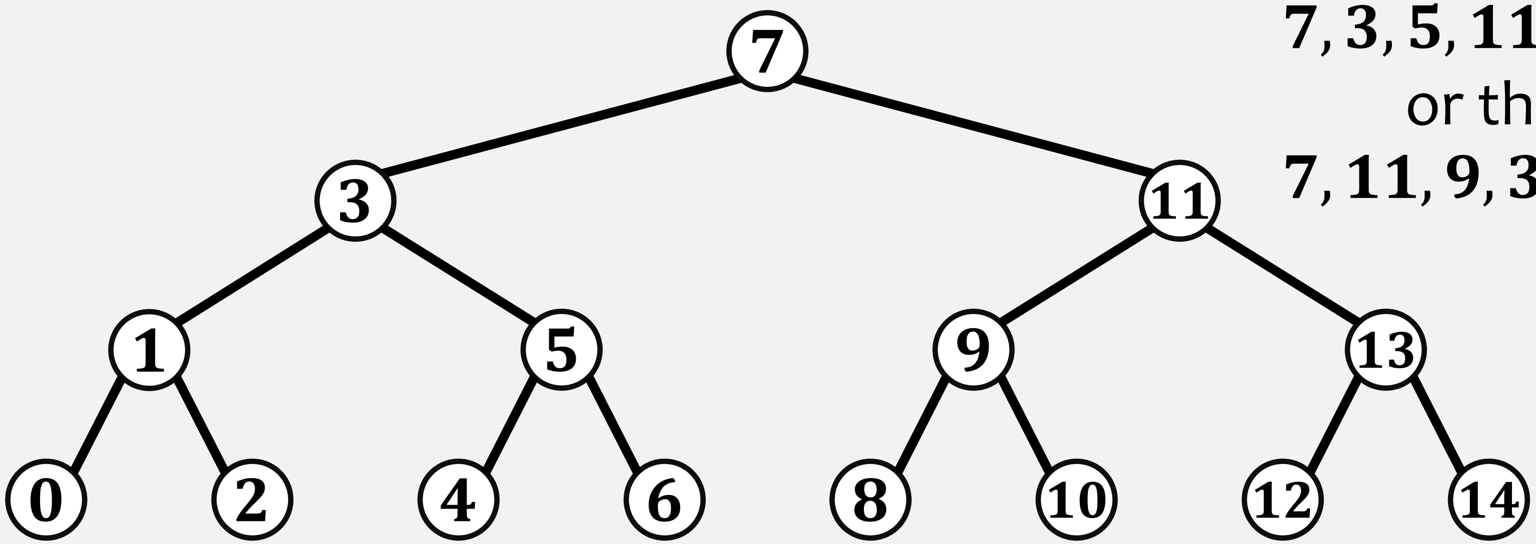
Alina Shaikhet

Let’s build a Tree

We want to create a binary search tree containing the integers $\mathbf{0,\ldots,14}$.

- Start with an empty tree,
- Add elements to the tree one at a time.

Shape of the tree will depend on the order, in which we add elements to the tree.



Let's add elements in this order:

$\mathbf{7, 3, 5, 11, 13, 1, 4, 9, 12, 2, 6, 14, 0, 10, 8}$

or this:

$\mathbf{7, 11, 9, 3, 1, 2, 8, 5, 10, 4, 13, 0, 12, 6, 14}$

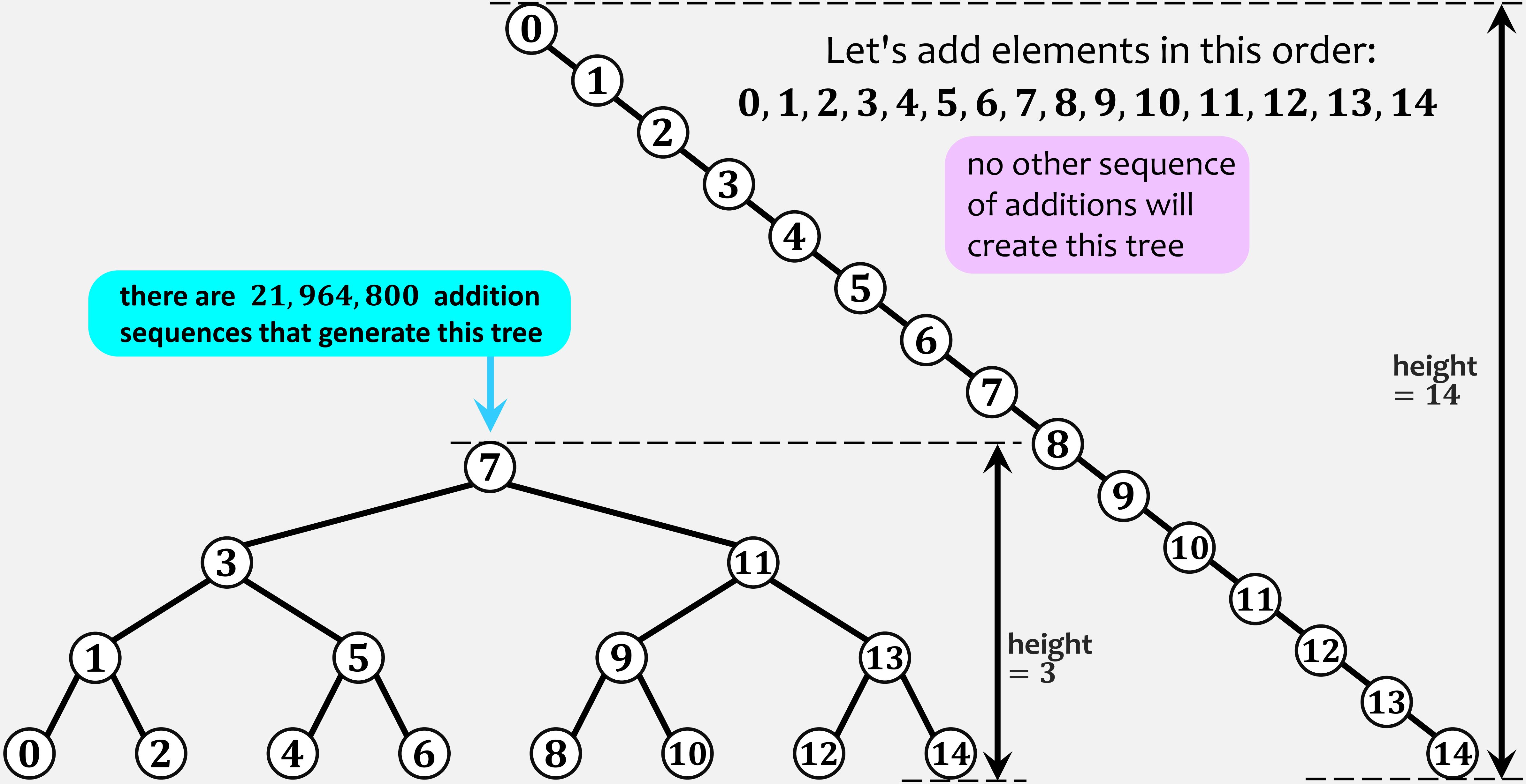
Two binary search trees containing the integers $\mathbf{0, \ldots, 14}$

Let's add elements in this order:

$\mathbf{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14}$

no other sequence of additions will create this tree

there are $\mathbf{21,964,800}$ addition sequences that generate this tree



Random Binary Search Tree

- Take a **random permutation** of $\boldsymbol{n}$ distinct values (ranks) (for example $\mathbf{0}, \dots, \boldsymbol{n}-\mathbf{1}$).
- Insert elements according to their ranks one at a time into a binary search tree.

In how many ways can we permute $\boldsymbol{n}$ elements?

$$\boldsymbol{n}! = \mathbf{1} \times \mathbf{2} \times \mathbf{3} \times \cdots \times (\boldsymbol{n}-\mathbf{1}) \times \boldsymbol{n}$$

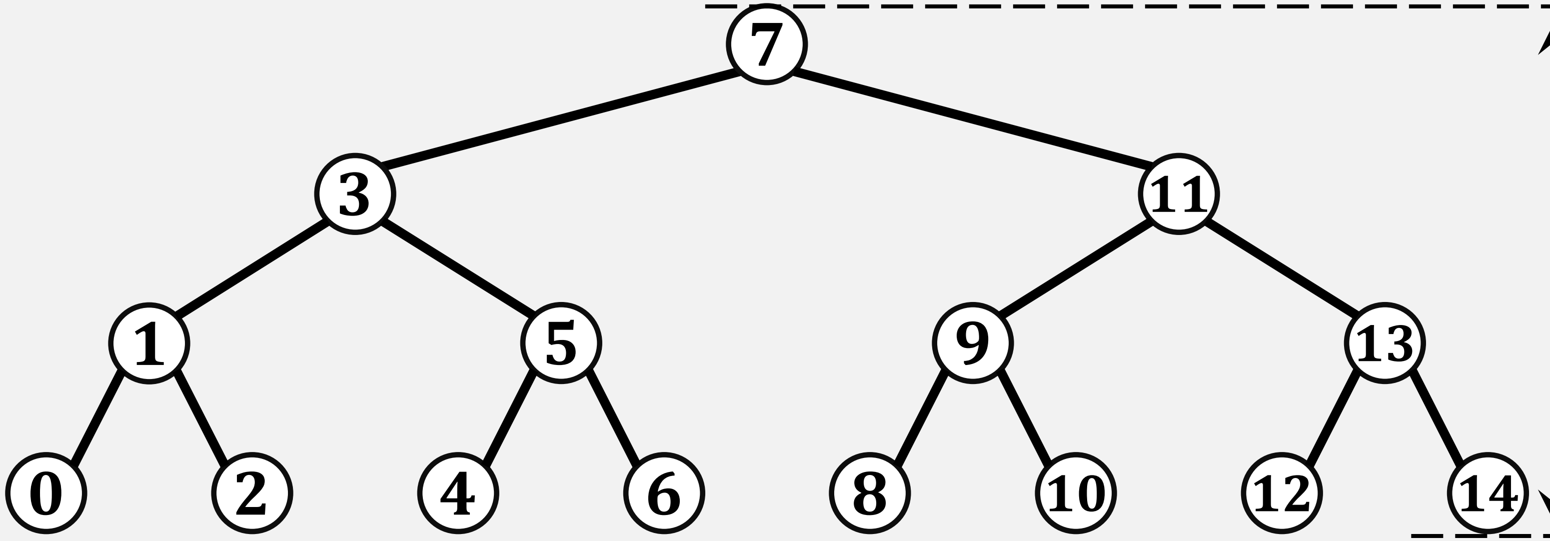
By **random permutation** we mean that each of the possible $\boldsymbol{n}!$ permutations (orderings) of $\mathbf{0}, \dots, \boldsymbol{n}-\mathbf{1}$ is equally likely,

So, the probability of obtaining any particular permutation is $\dfrac{\mathbf{1}}{\boldsymbol{n}!}$

Two binary search trees containing the integers $\mathbf{0, \ldots, 14}$

$$\frac{21,964,800}{1,307,674,368,000} \approx 0.0000168$$

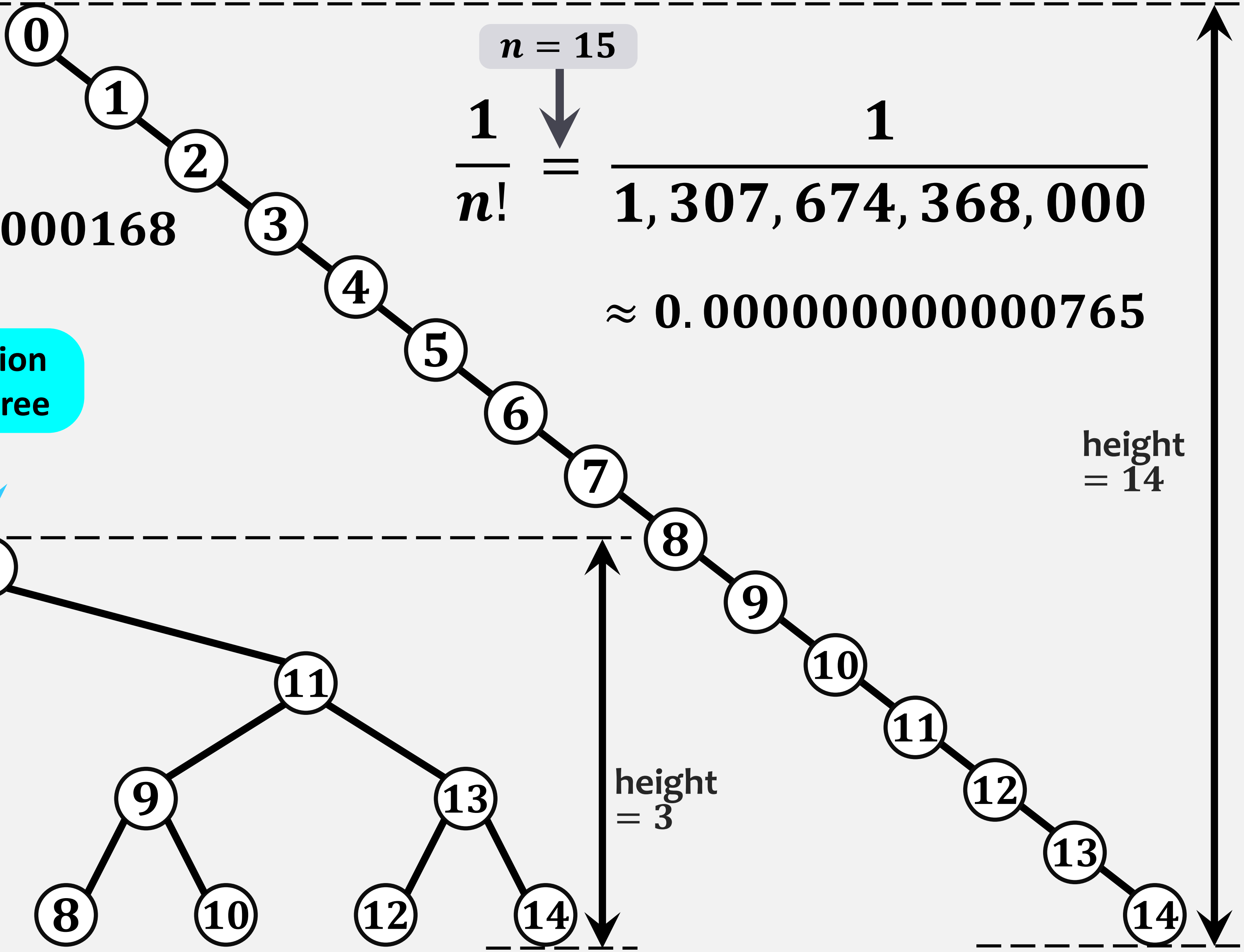
there are $21,964,800$ addition sequences that generate this tree



$n = 15$

$$\frac{1}{n!} = \frac{1}{1,307,674,368,000}$$

$$\approx 0.000000000000765$$



Alina Shaikhet – COMP 2402 – Carleton University

Lemma 7.1

In a random binary search tree with n nodes, the expected length of the **search path** for any value x (whether x is stored in the tree or not) is **at most**

$$\mathbf{2\ln n + 2 \approx 1.38\log_2 n + 2}$$

Recall: The expected length of a search path in a SkipList is at most $\mathbf{2\log_2 n + O(1)}$.

This result is with respect to the **random permutation** used to create the random binary search tree. We cannot maintain this result under add(x) and remove(x) operations.

The problem with random binary search trees is that **they are not dynamic**.

Random binary search trees don't support the add(x) or remove(x) operations.

So, we cannot use them to implement the **SSet** interface.

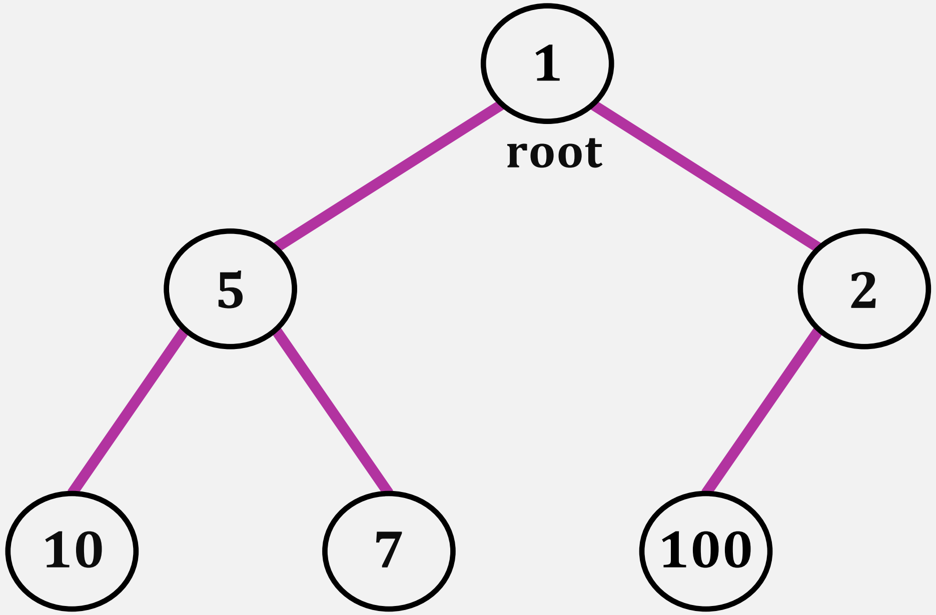
Heap

Heap is a binary tree where each node $\boldsymbol{u}$ has a **priority** $\boldsymbol{u.p}$

(Min-)Heap Property:

For each node $\boldsymbol{u}$ except the root:

$$\boldsymbol{u}.\text{parent}.\boldsymbol{p} < \boldsymbol{u}.\boldsymbol{p}$$



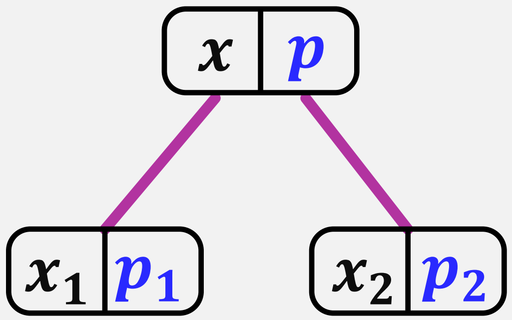
The node with **minimum priority** has to be the **root**.

You cannot effectively search for things in heaps. They are not designed for this.

Treap

A **Treap** is a binary tree $\boldsymbol{T}$ where each node has:

- a value $\boldsymbol{x}$ **(key)**
- a priority $\boldsymbol{p}$



$\boldsymbol{T}$ is a binary search tree with respect to the $\boldsymbol{x}$ values: $\boldsymbol{x_1 < x < x_2}$

$\boldsymbol{T}$ is a heap with respect to the $\boldsymbol{p}$ values. (each node has a priority smaller than that of its two children): $\boldsymbol{p < p_1}$ and $\boldsymbol{p < p_2}$

Priority values in a Treap are **unique** and **assigned at random.**

Treap

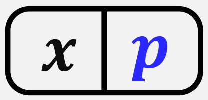
$\boldsymbol{x}$	$\boldsymbol{p}$

- Node with the smallest priority is a root $(\boldsymbol{r})$.
- All nodes with keys **smaller** than $\boldsymbol{r}.\boldsymbol{x}$ are stored in the subtree rooted at $\boldsymbol{r}$.left.
  All nodes with keys **larger** than $\boldsymbol{r}.\boldsymbol{x}$ are stored in the subtree rooted at $\boldsymbol{r}$.right.
- Repeat recursively

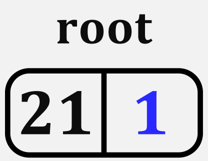
root

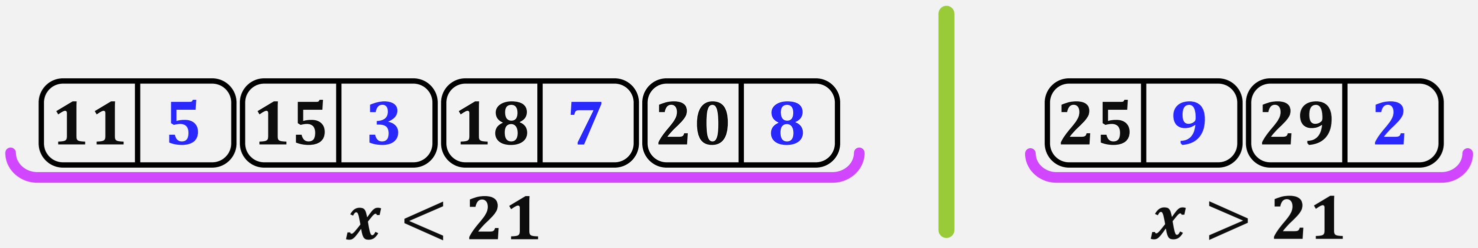
11	5	15	3	18	7	20	8	21	1	25	9	29	2

Treap



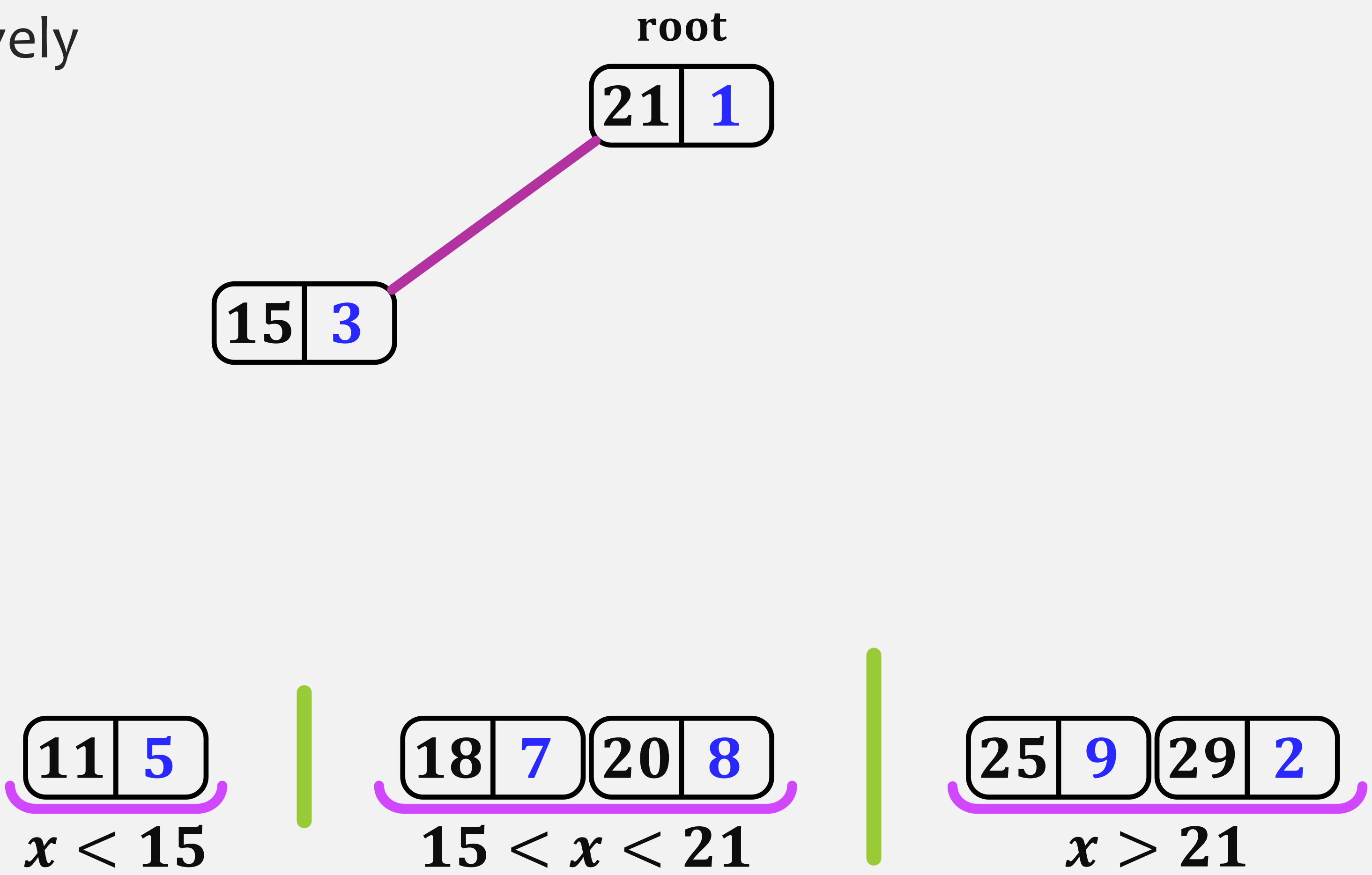
- Node with the smallest priority is a root (r).
- All nodes with keys **smaller** than $r.x$ are stored in the subtree rooted at r.left.
  All nodes with keys **larger** than $r.x$ are stored in the subtree rooted at r.right.
- Repeat recursively



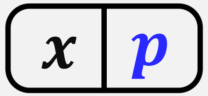


Treap

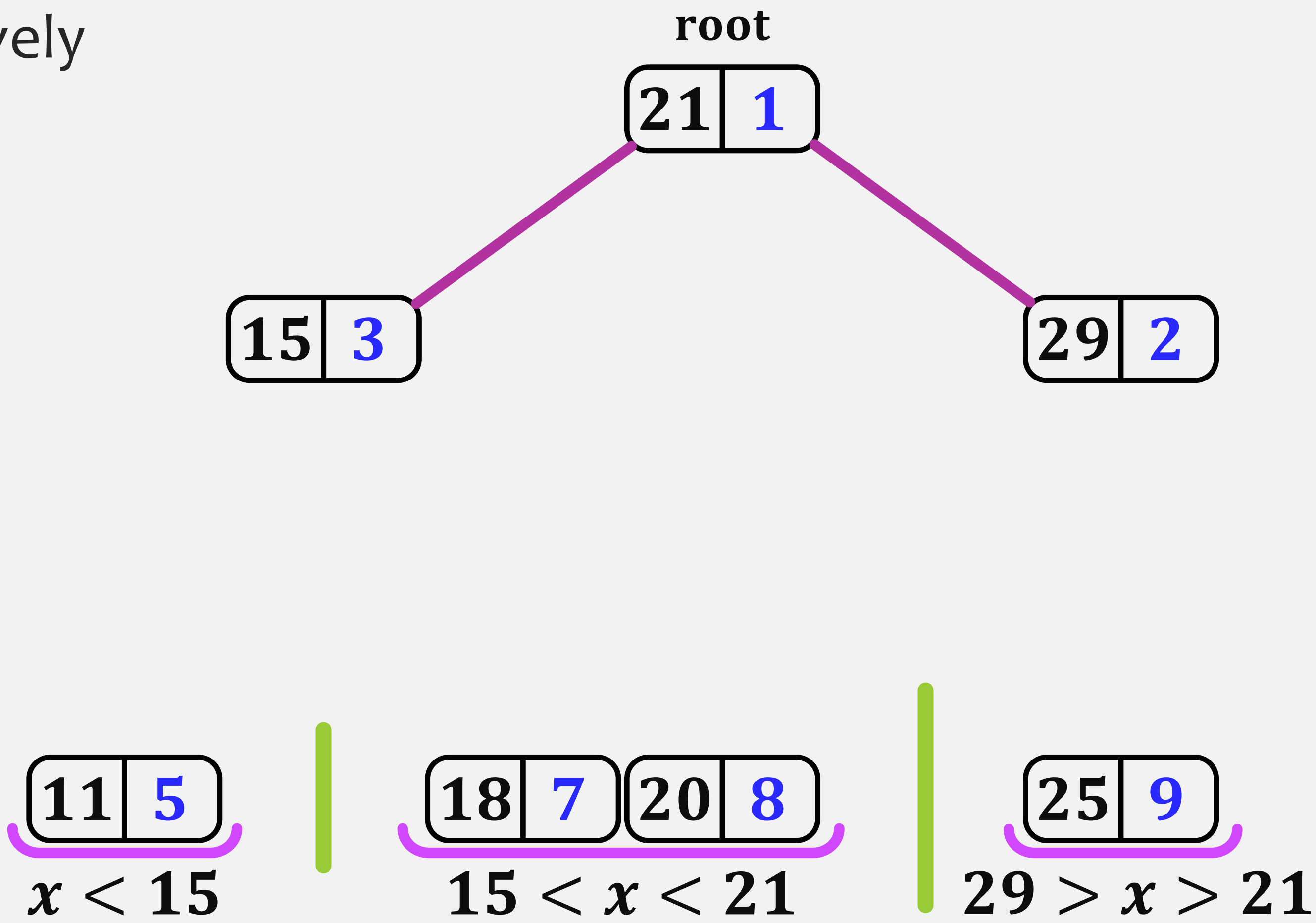
- Node with the smallest priority is a root (r).
- All nodes with keys **smaller** than $r.x$ are stored in the subtree rooted at r.left.
  All nodes with keys **larger** than $r.x$ are stored in the subtree rooted at r.right.
- Repeat recursively



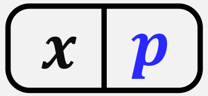
Treap



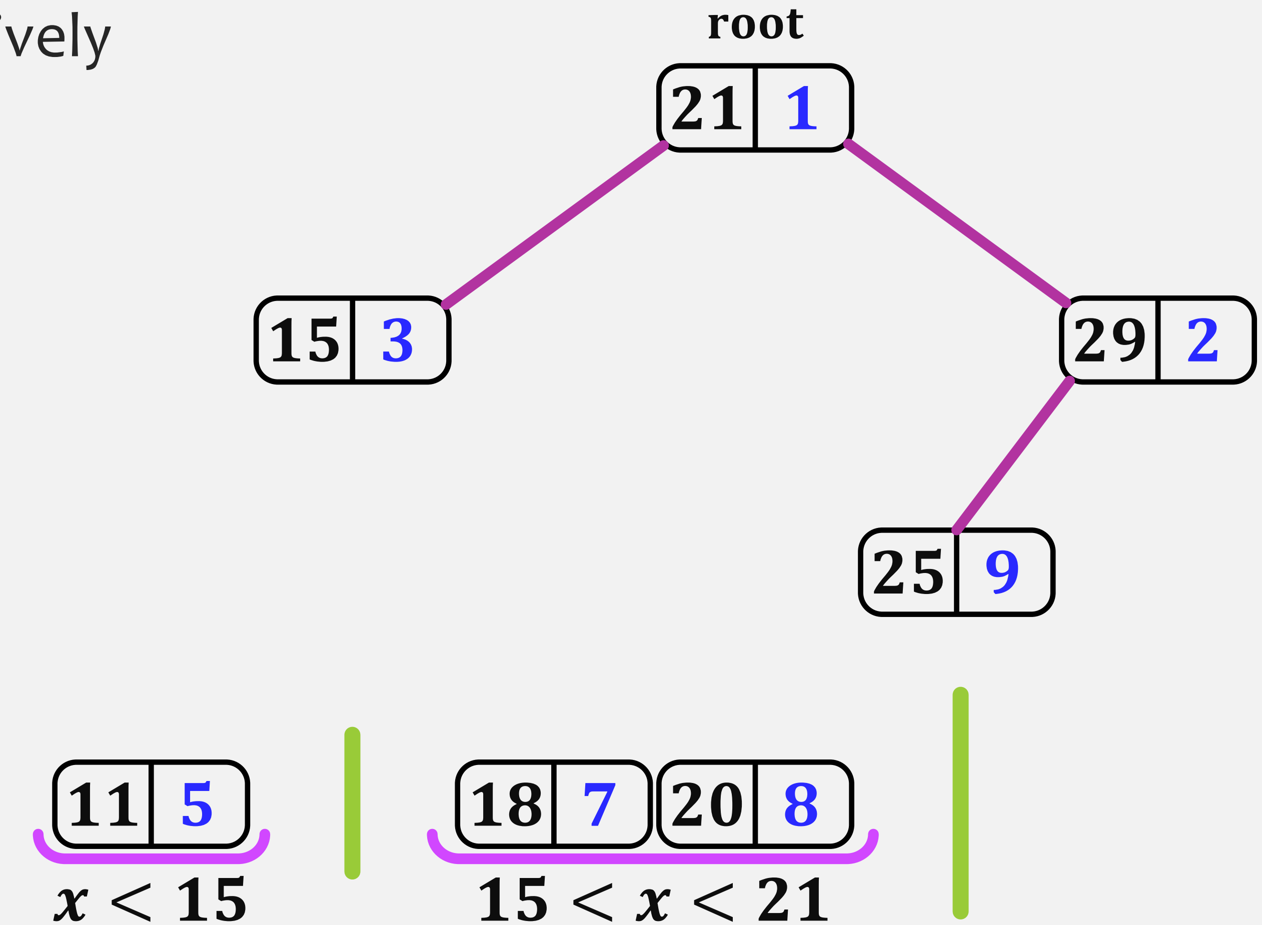
- Node with the smallest priority is a root (r).
- All nodes with keys **smaller** than $r.x$ are stored in the subtree rooted at r.left.
  All nodes with keys **larger** than $r.x$ are stored in the subtree rooted at r.right.
- Repeat recursively



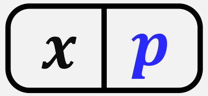
Treap



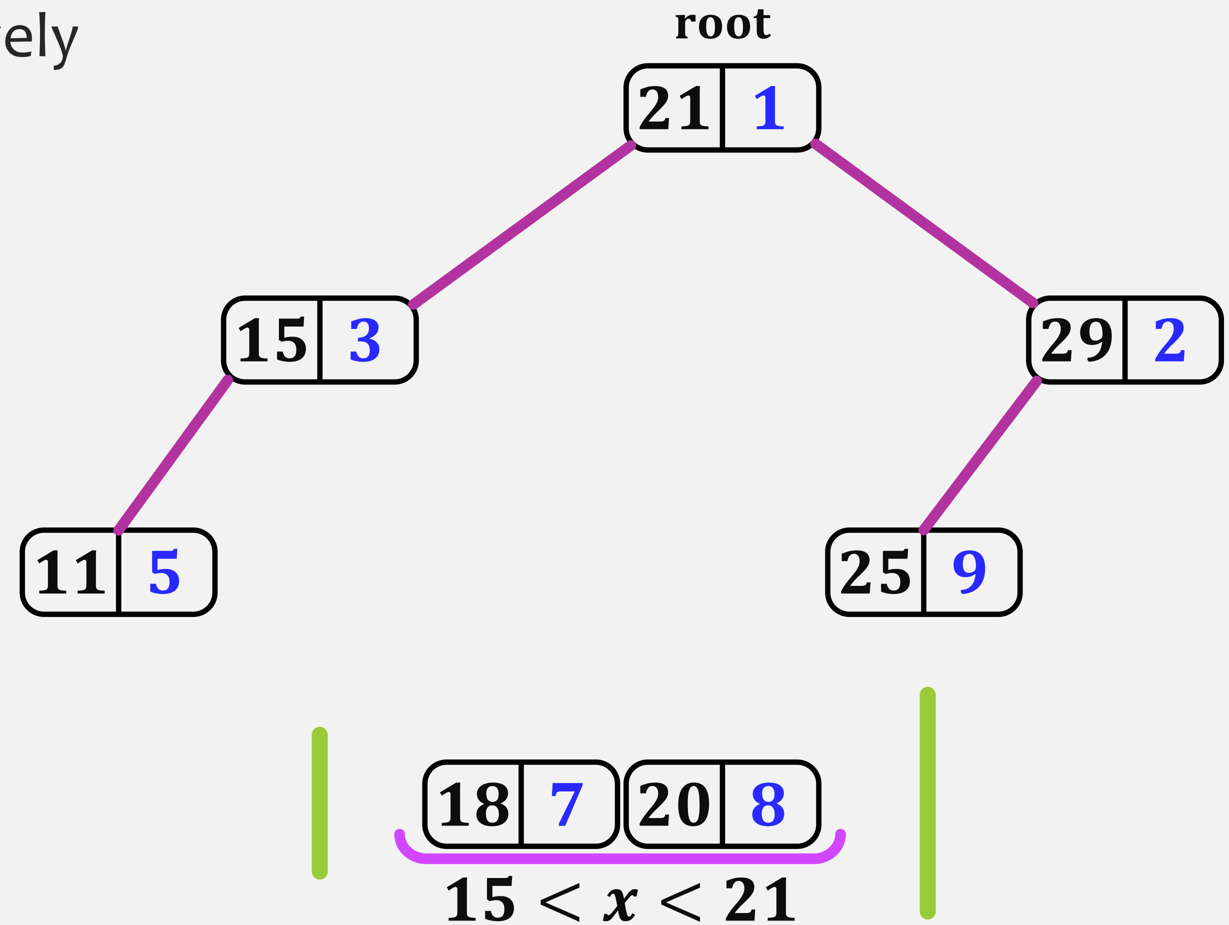
- Node with the smallest priority is a root $(\boldsymbol{r})$.
- All nodes with keys **smaller** than $\boldsymbol{r.x}$ are stored in the subtree rooted at $\boldsymbol{r}$.left.
  All nodes with keys **larger** than $\boldsymbol{r.x}$ are stored in the subtree rooted at $\boldsymbol{r}$.right.
- Repeat recursively



Treap

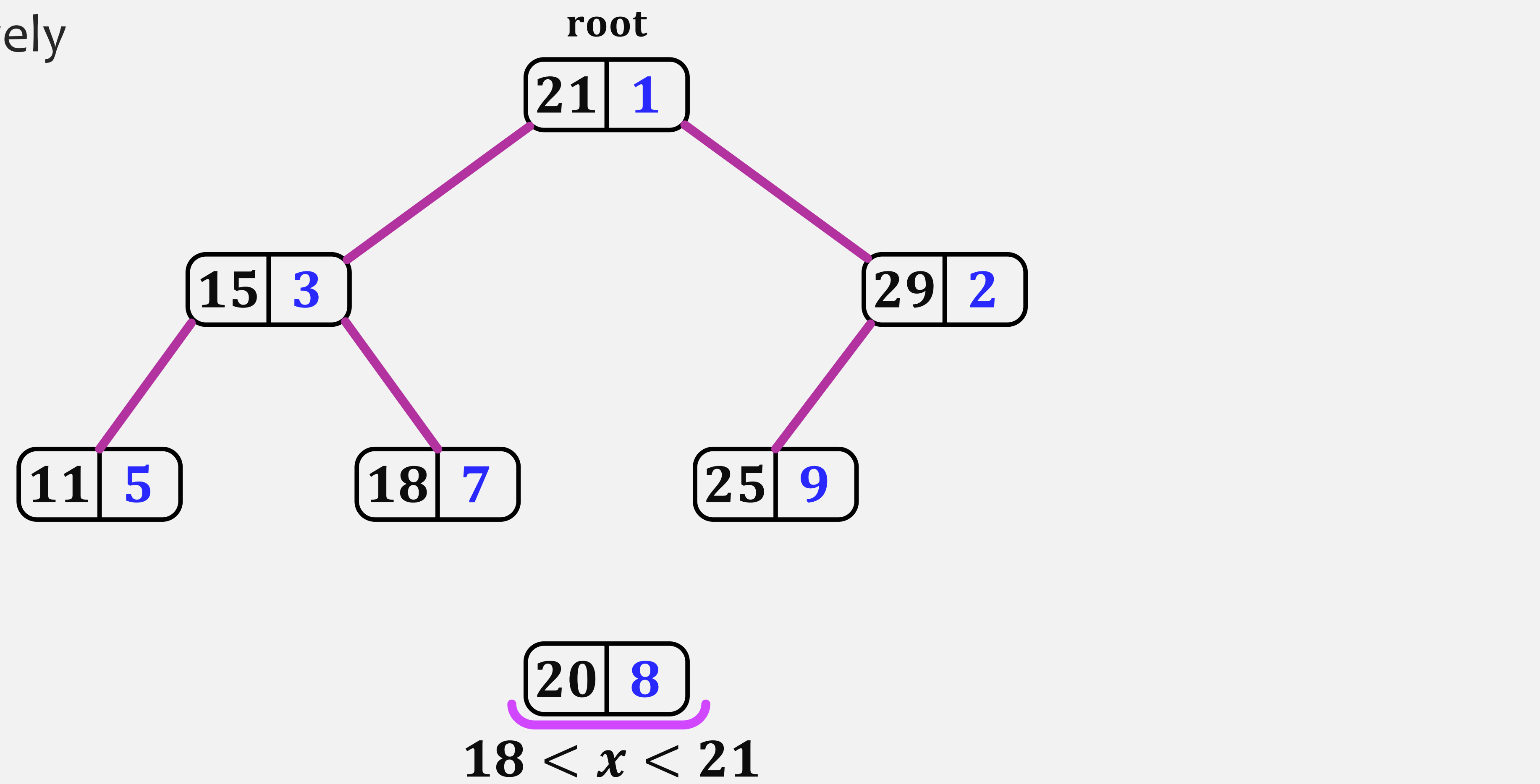


- Node with the smallest priority is a root (r).
- All nodes with keys **smaller** than $r.x$ are stored in the subtree rooted at r.left.
  All nodes with keys **larger** than $r.x$ are stored in the subtree rooted at r.right.
- Repeat recursively

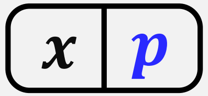


Treap

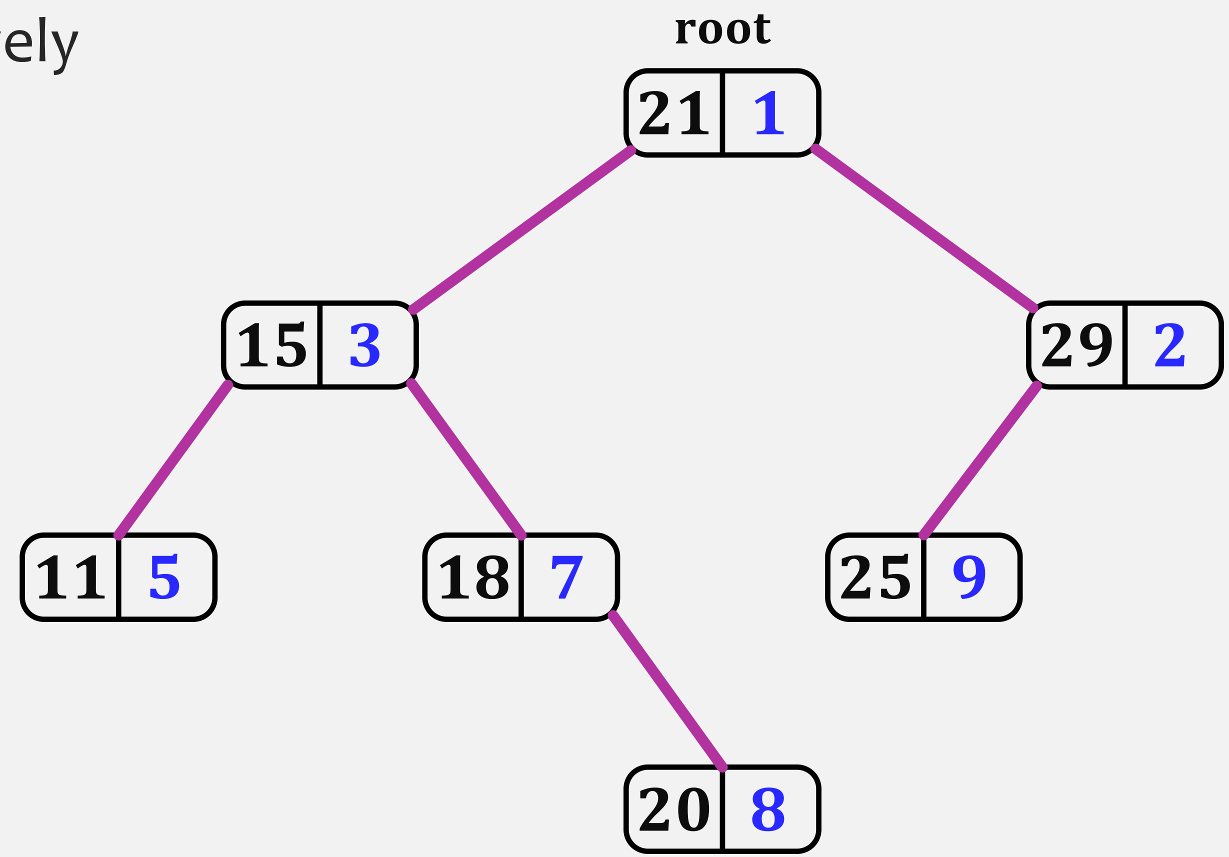
- Node with the smallest priority is a root $(\boldsymbol{r})$.
- All nodes with keys **smaller** than $\boldsymbol{r.x}$ are stored in the subtree rooted at $\boldsymbol{r}$.left.
  All nodes with keys **larger** than $\boldsymbol{r.x}$ are stored in the subtree rooted at $\boldsymbol{r}$.right.
- Repeat recursively



Treap

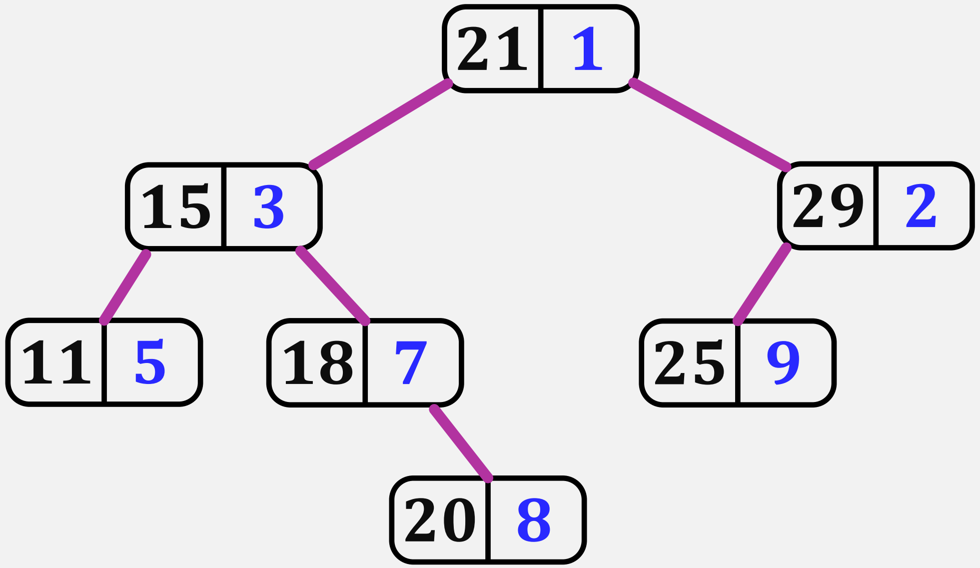


- Node with the smallest priority is a root $(\boldsymbol{r})$.
- All nodes with keys **smaller** than $\boldsymbol{r}.\boldsymbol{x}$ are stored in the subtree rooted at $\boldsymbol{r}$.left.
  All nodes with keys **larger** than $\boldsymbol{r}.\boldsymbol{x}$ are stored in the subtree rooted at $\boldsymbol{r}$.right.
- Repeat recursively

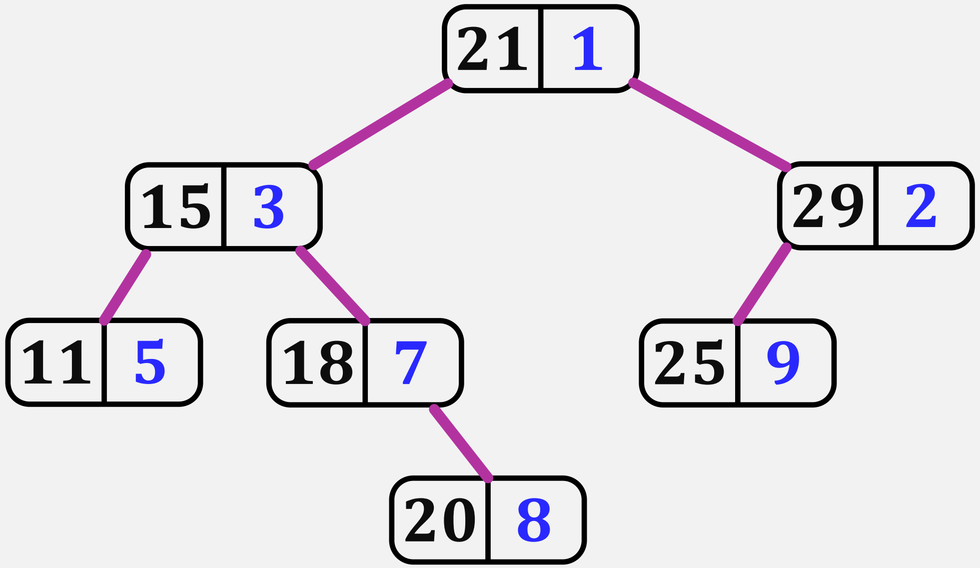


Alina Shaikhet – COMP 2402 – Carleton University

Treap

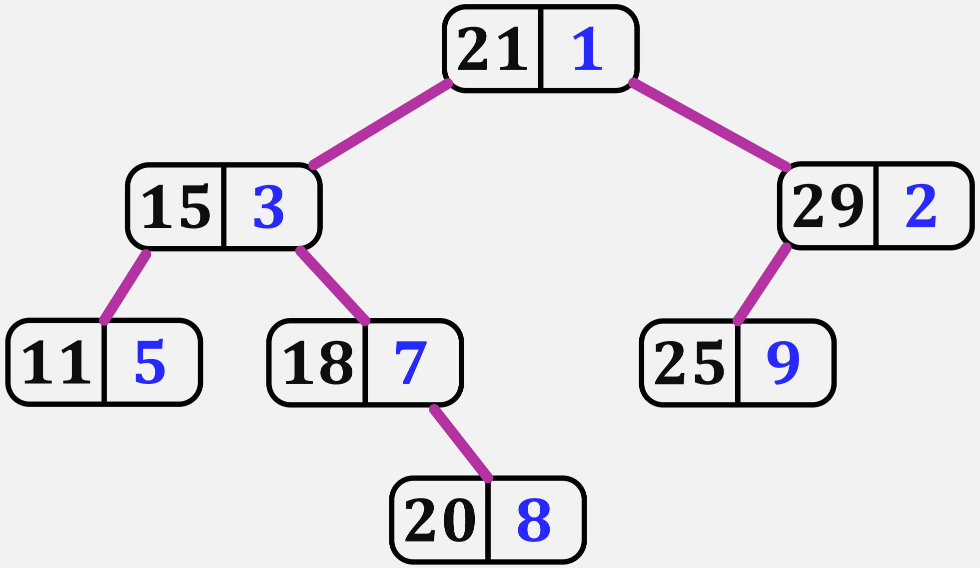


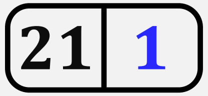
Treap


21 1
15 3
29 2
11 5
18 7
25 9
20 8


sorted by priority
21 1
29 2
15 3
11 5
18 7
20 8
25 9

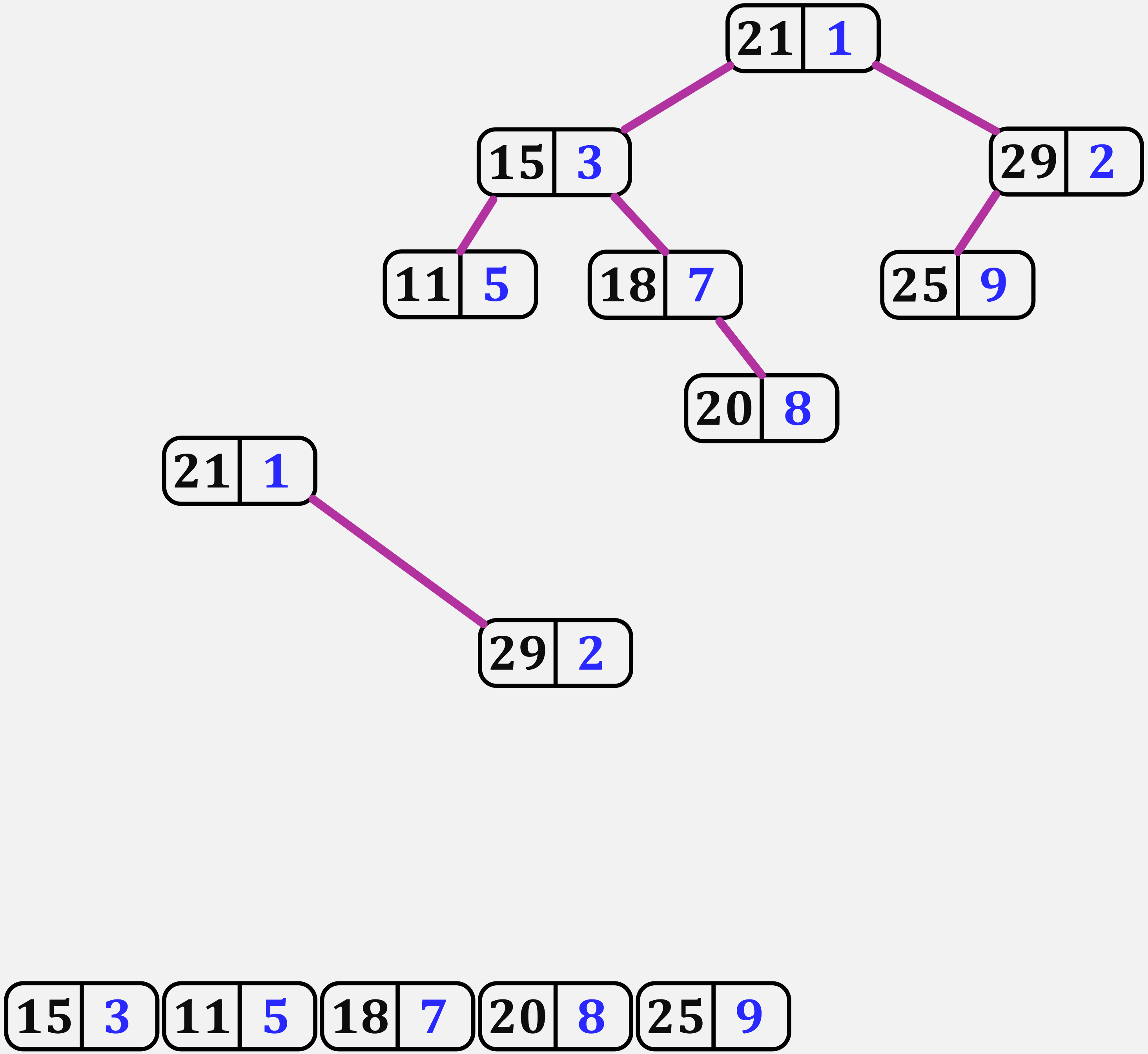
Treap





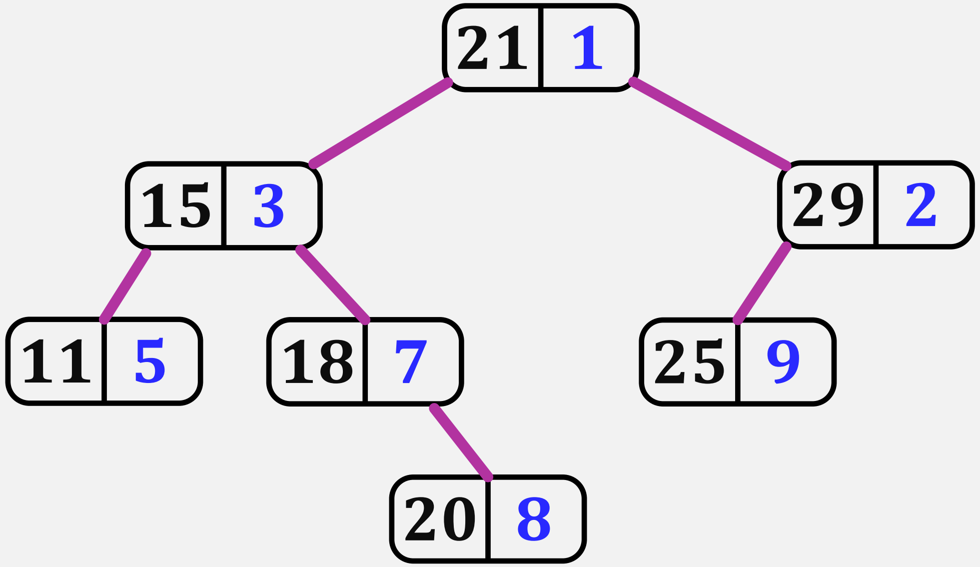
sorted by priority

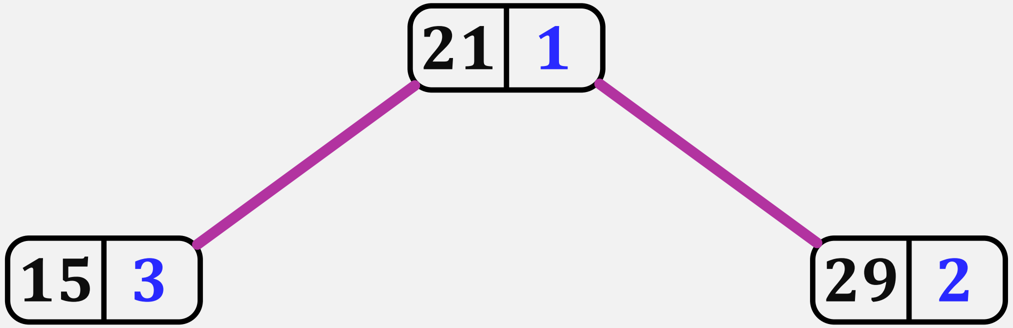
Treap



sorted by priority

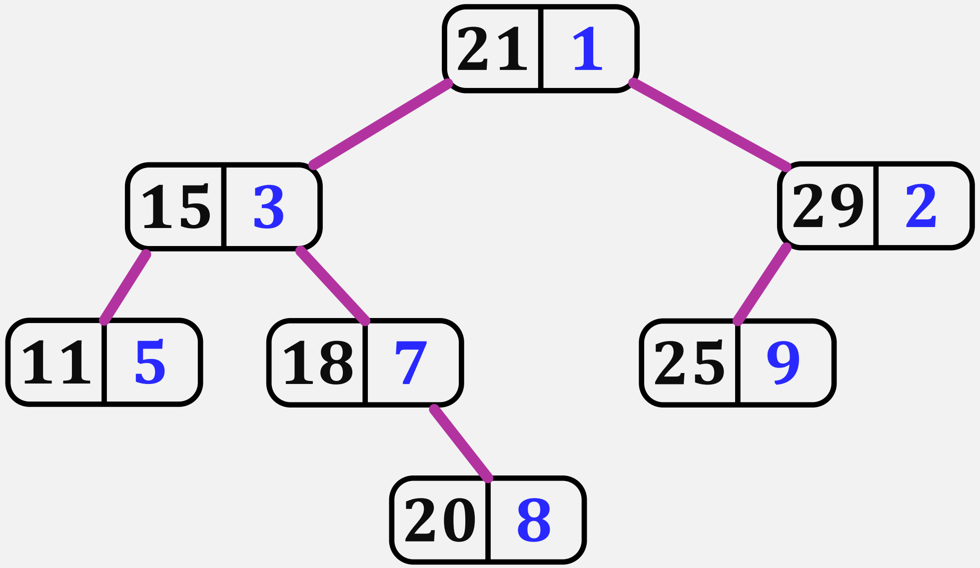
Treap

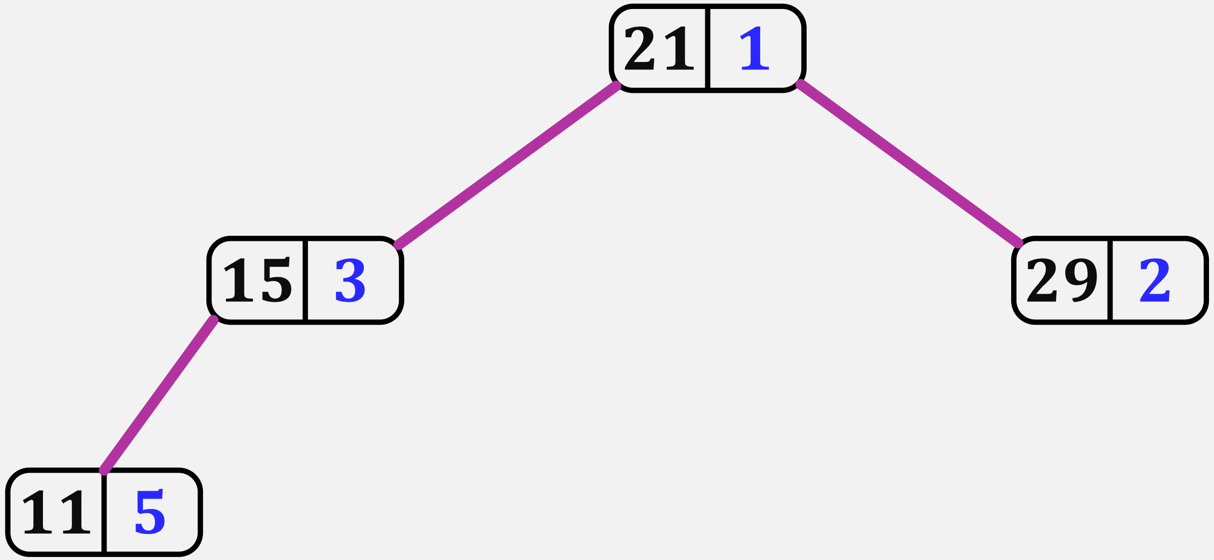




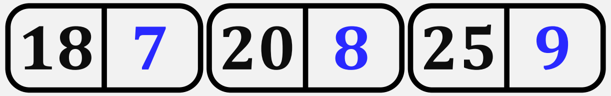
sorted by priority

Treap

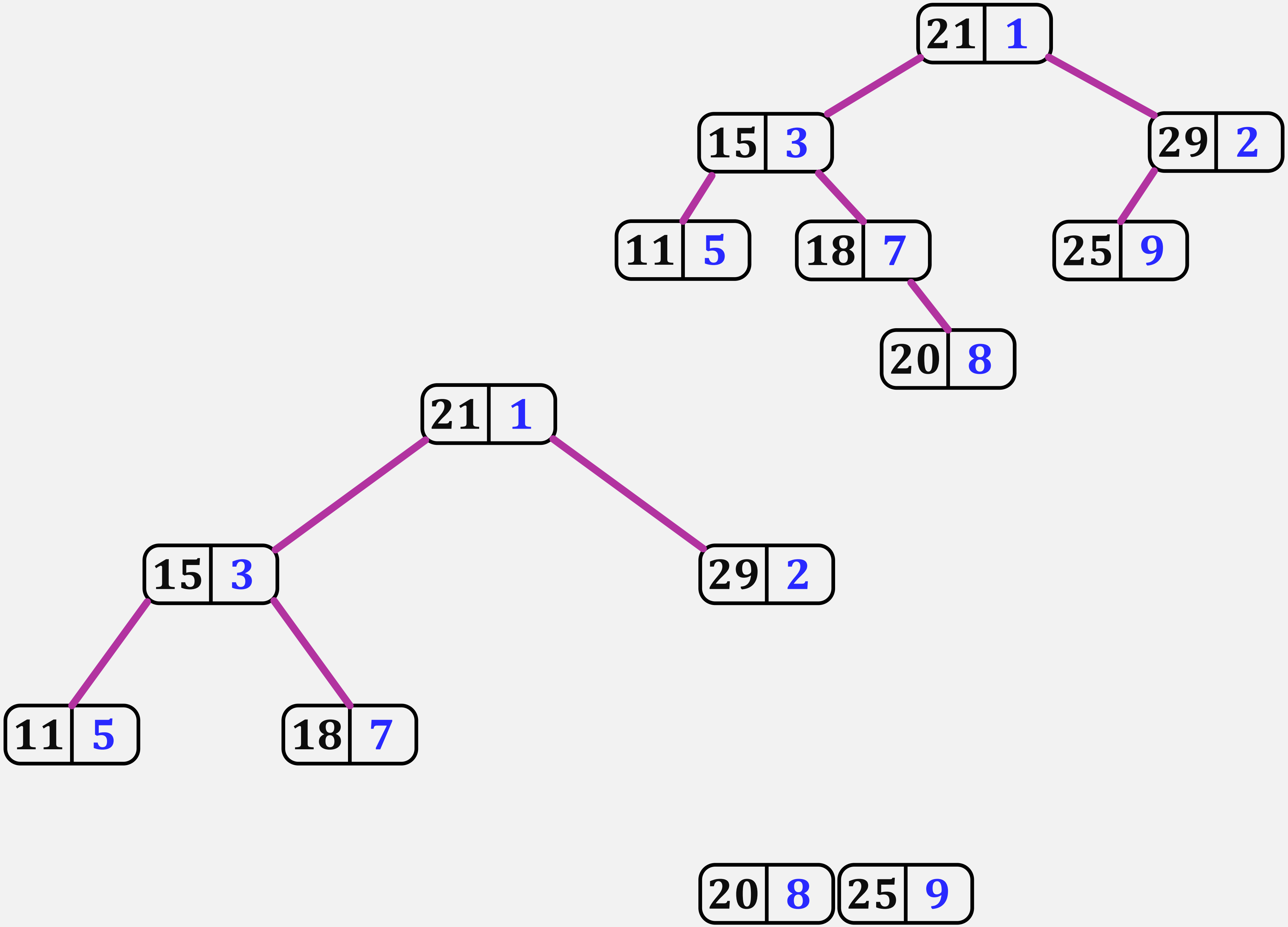




sorted by priority

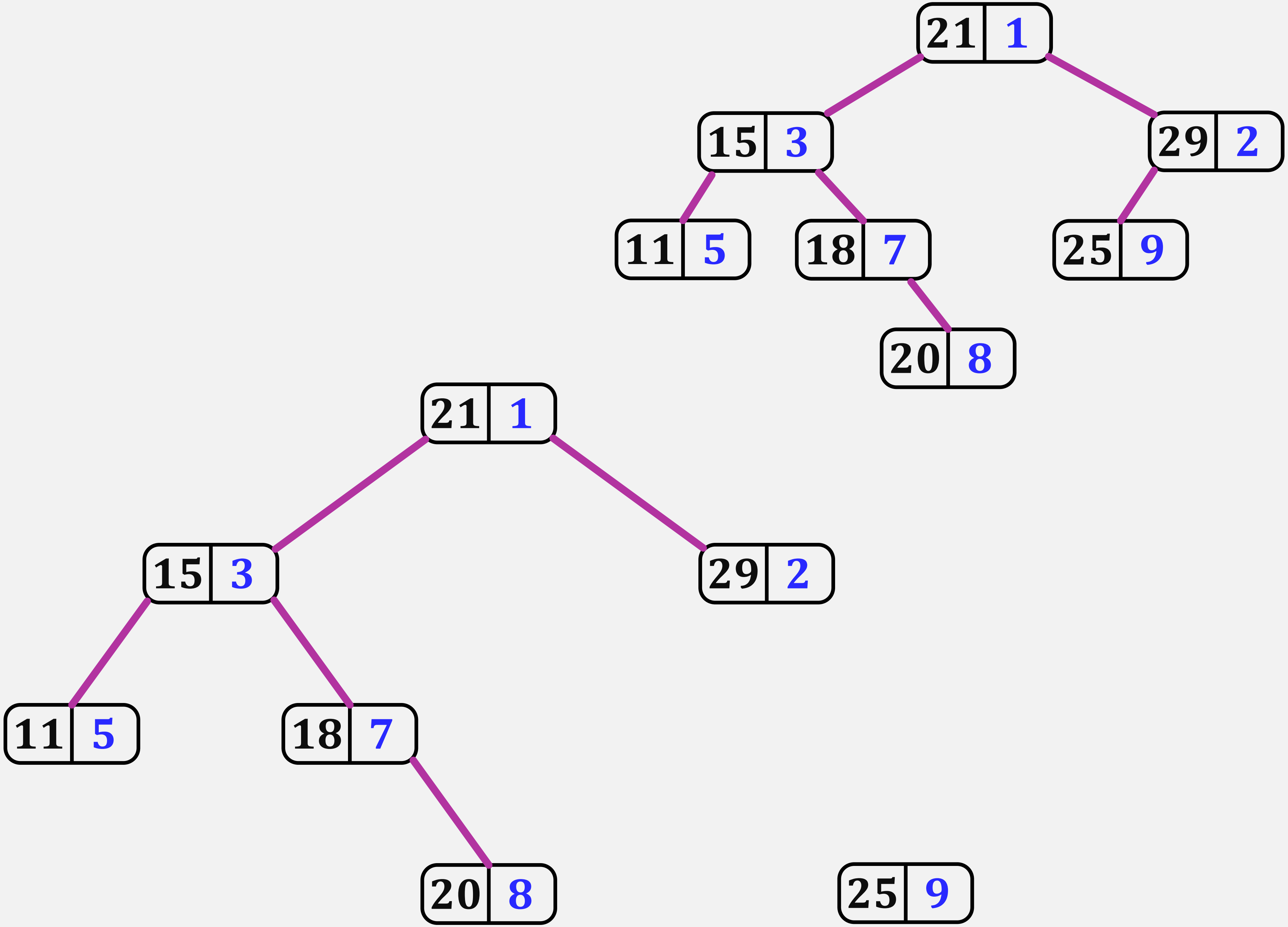


Treap

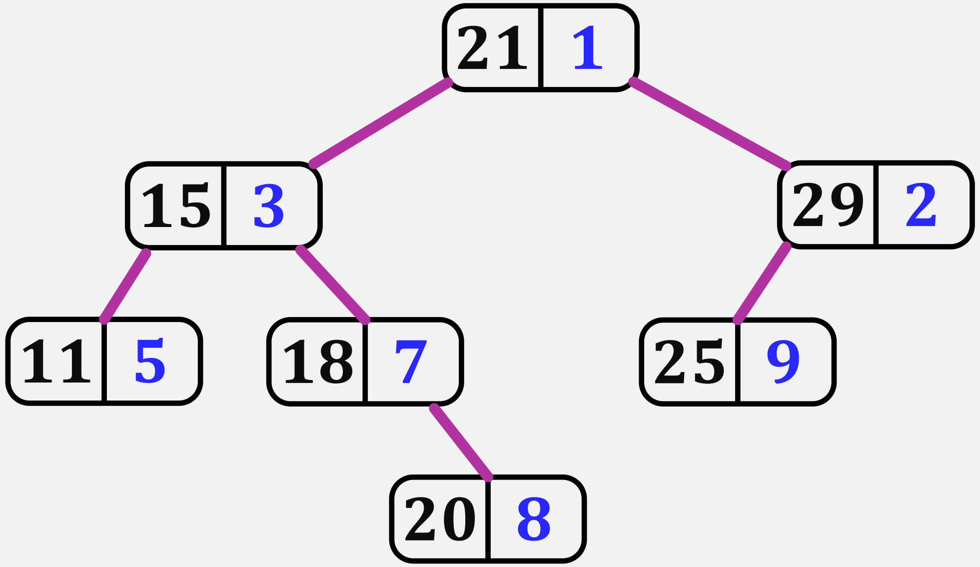


sorted by priority

Treap

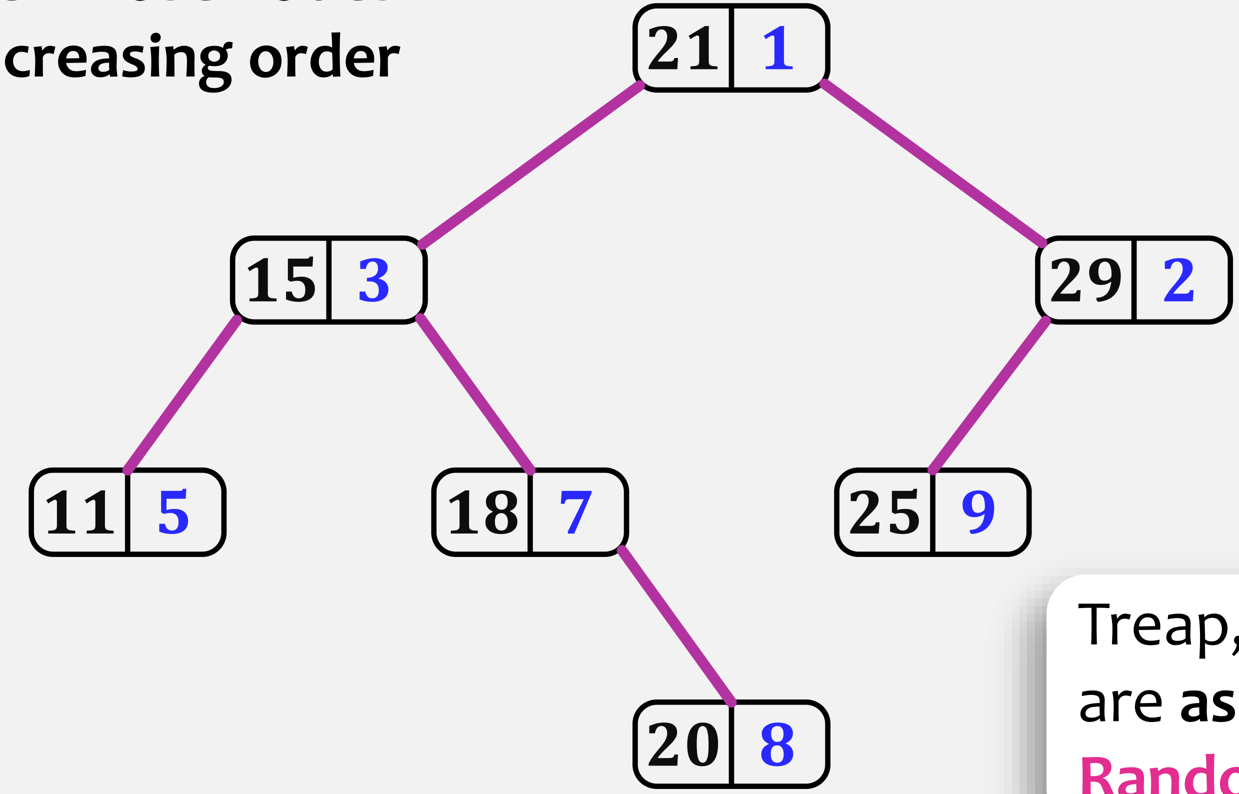


Treap



Alternative definition of a Treap:

We can think of a Treap as a BinarySearchTree whose nodes were added in **increasing order of priority**.



Treap, where the **priorities** are **assigned at random**, is a **Random Binary Search Tree**

Lemma 7.2

In a Treap (where the **priorities** are **assigned at random)** with n nodes, the expected length of the **search path** for any value x (whether x is stored in the treap or not) is **at most**

$$\mathbf{2\ln n + 2 \approx 1.38\log_2 n + 2}$$

Treap: SSet implementation

- add($\boldsymbol{x}$),
- remove($\boldsymbol{x}$),
- find($\boldsymbol{x}$) – find the **smallest** value that is $\geq \boldsymbol{x}$.

Search in a Treap for $\boldsymbol{x}$. It will take $\boldsymbol{O}(\mathbf{log}\,\boldsymbol{n})$ **expected** time.

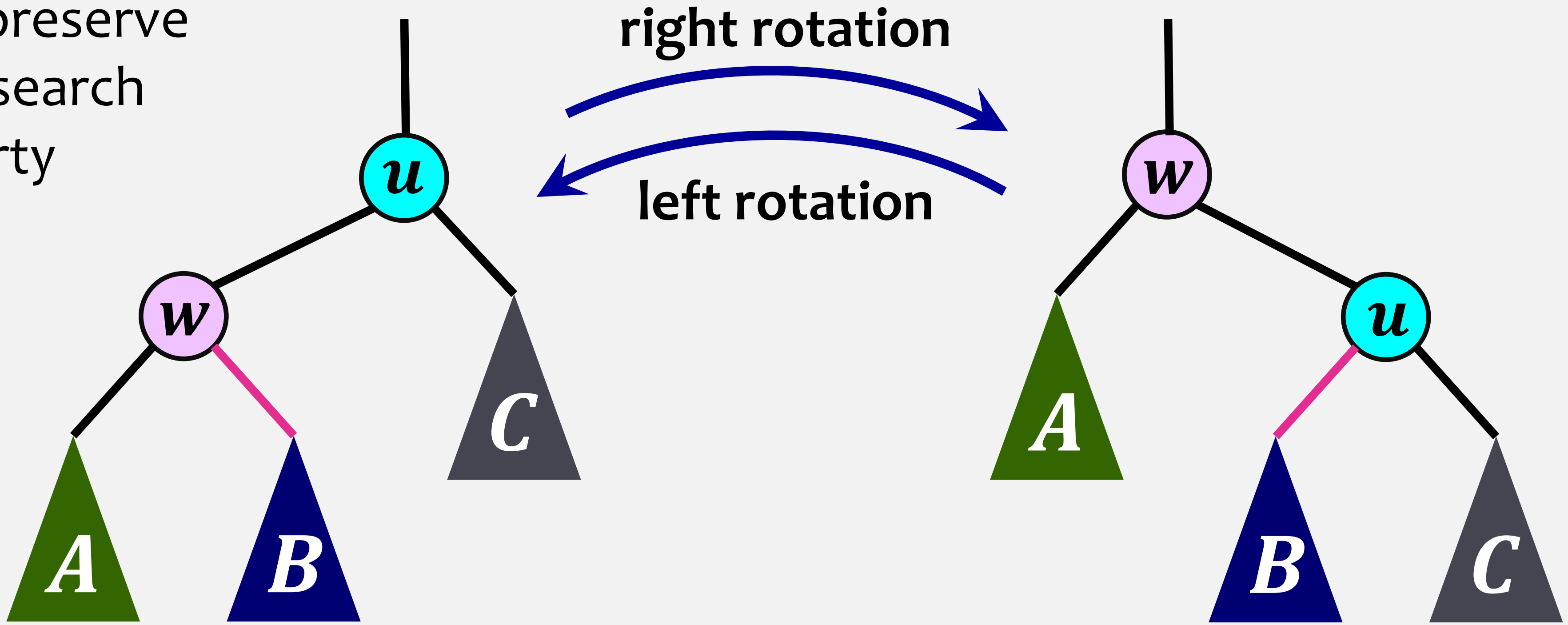
if we can maintain a random treap

To support the add($\boldsymbol{x}$) and remove($\boldsymbol{x}$) operations, a treap needs to perform **rotations** in order to maintain the heap property.

Rotation

the most important property of a rotation (right) is that the depth of $\boldsymbol{w}$ decreases by one while the depth of $\boldsymbol{u}$ increases by one.

Rotations preserve the binary search tree property

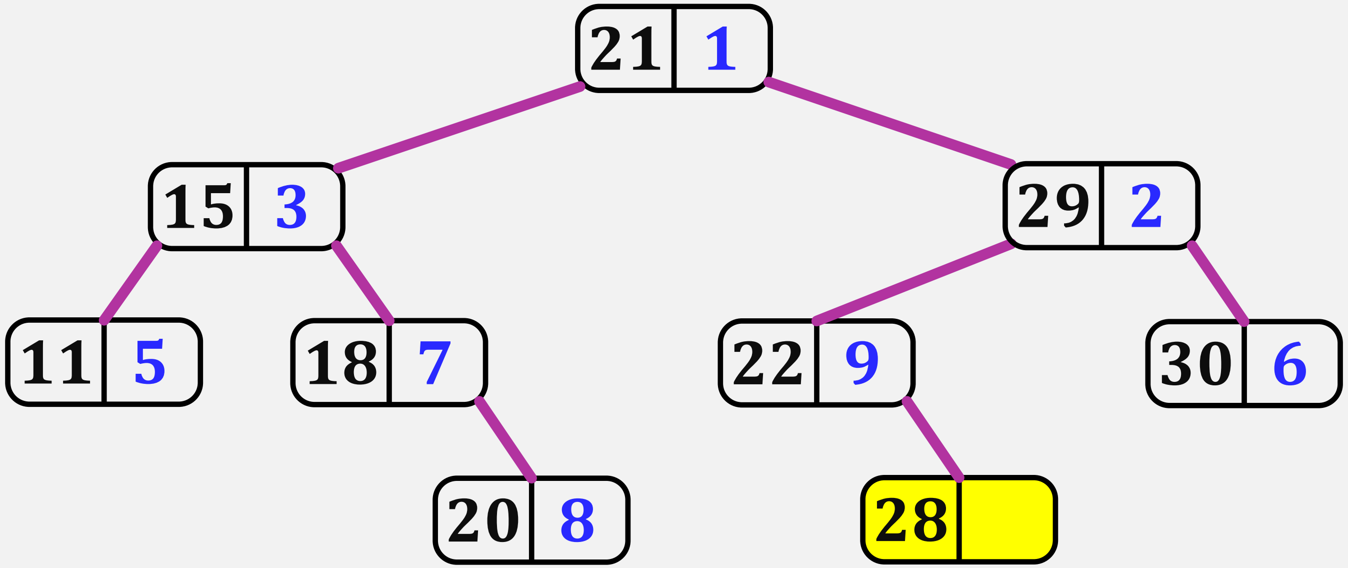


$$\boldsymbol{A} < \boldsymbol{w} < \boldsymbol{B} < \boldsymbol{u} < \boldsymbol{C}$$

Any of the trees $\boldsymbol{A}$, $\boldsymbol{B}$ or $\boldsymbol{C}$ can be empty.

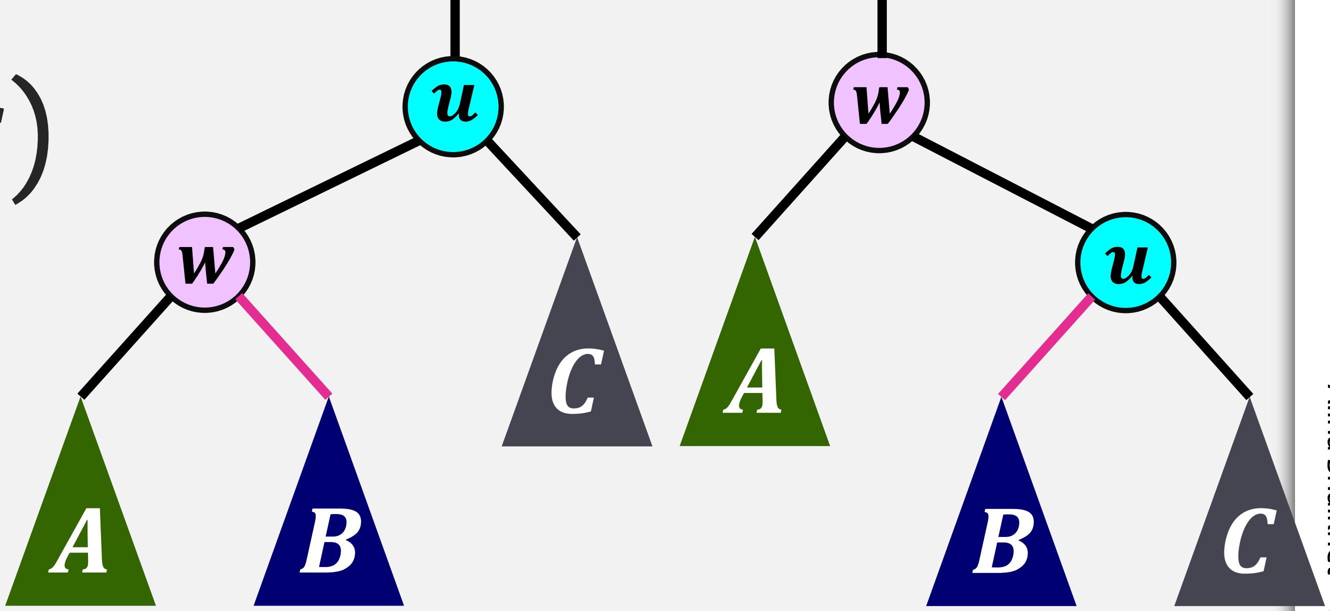
Treap: SSet – add($\boldsymbol{x}$)

- Search for $\boldsymbol{x}$ in the tree
- If it is not there, add $\boldsymbol{x}$ as a leaf: $\boldsymbol{q}$

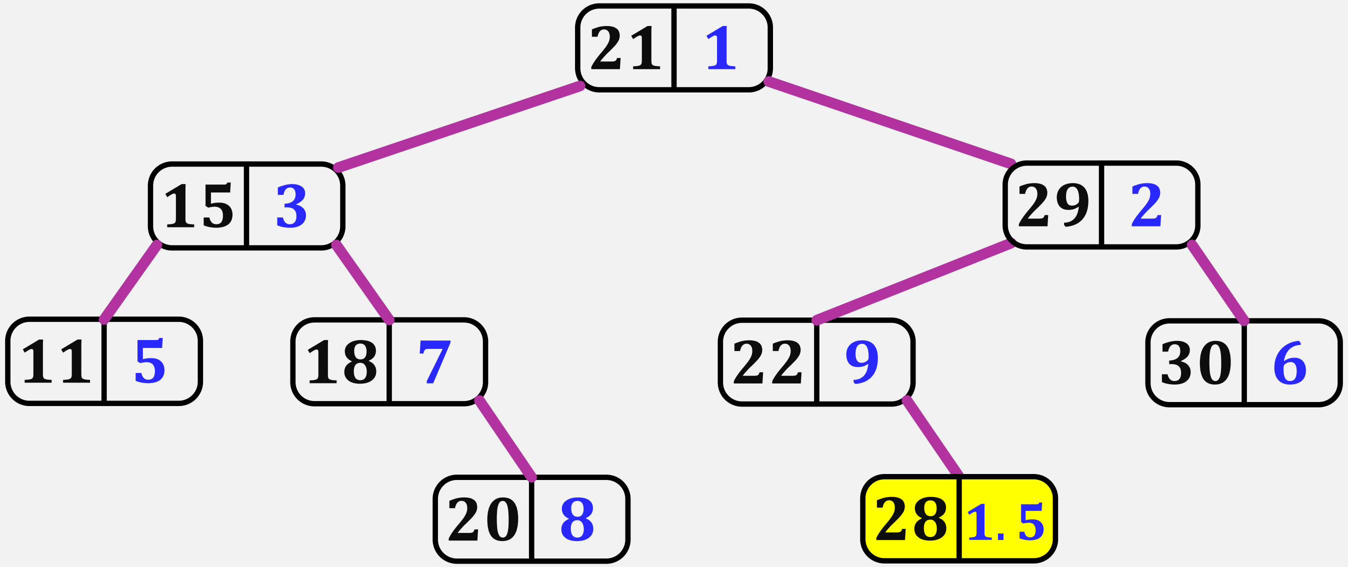


add(**28**)

Treap: SSet – add(x)



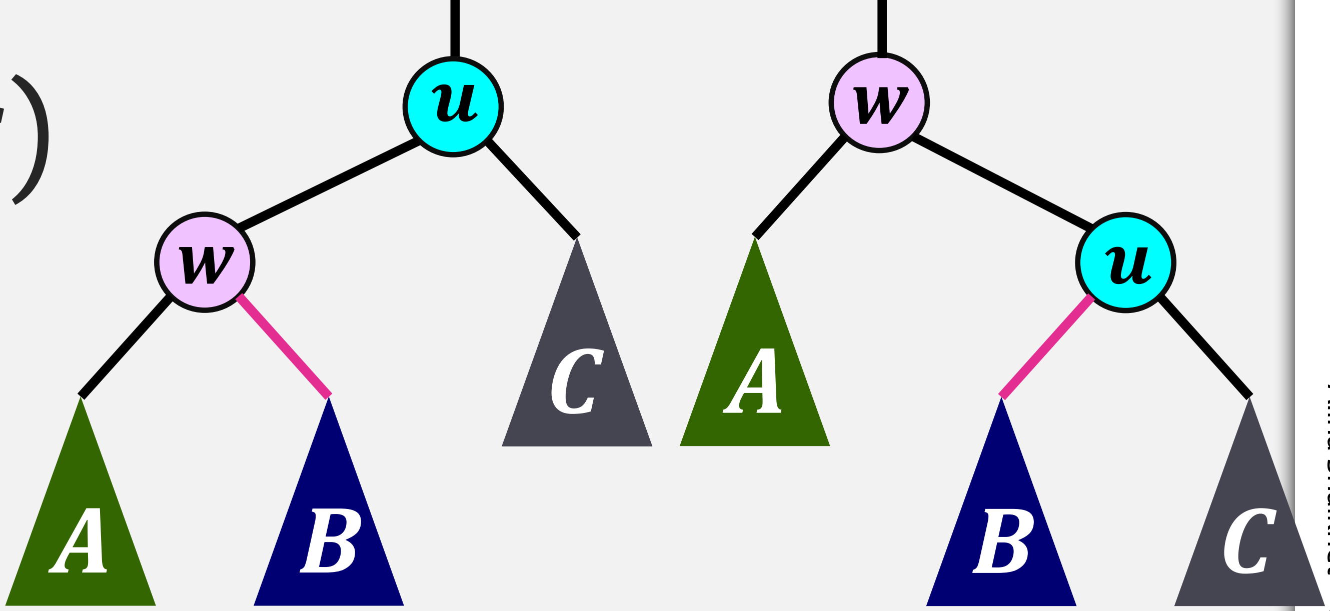
- Search for x in the tree
- If it is not there, add x as a leaf: q
- Give it a random priority: $q.p$
- As long $q.p < q.\text{parent}.p$ rotate it.

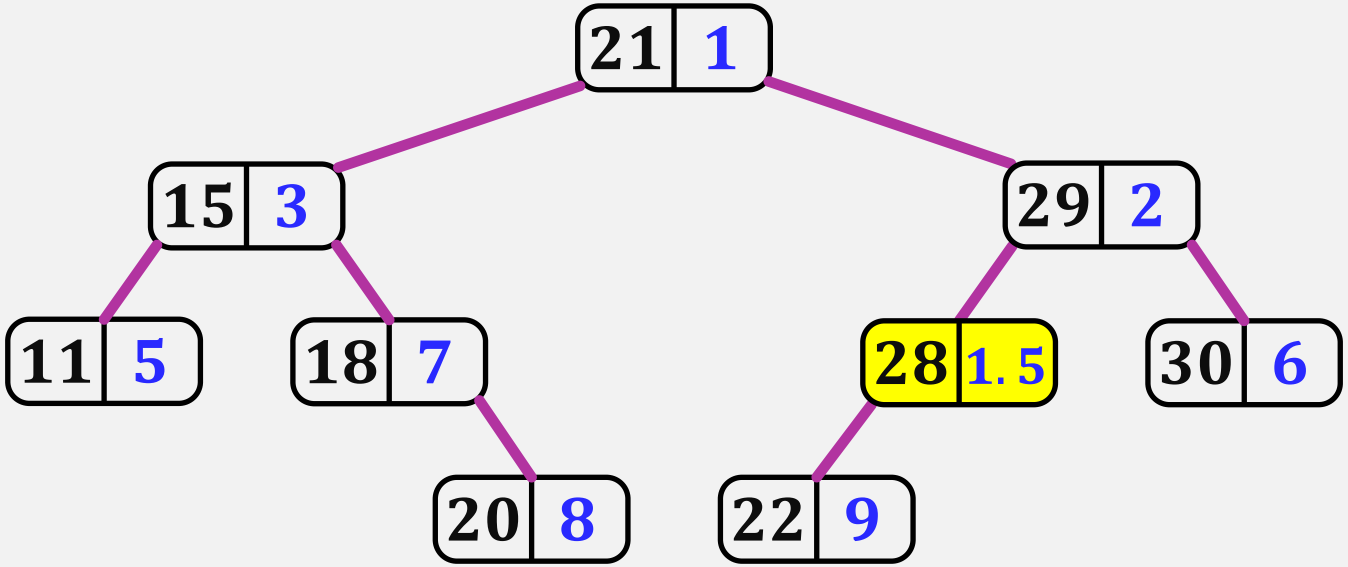


add(**28**)

Treap: SSet – add(x)

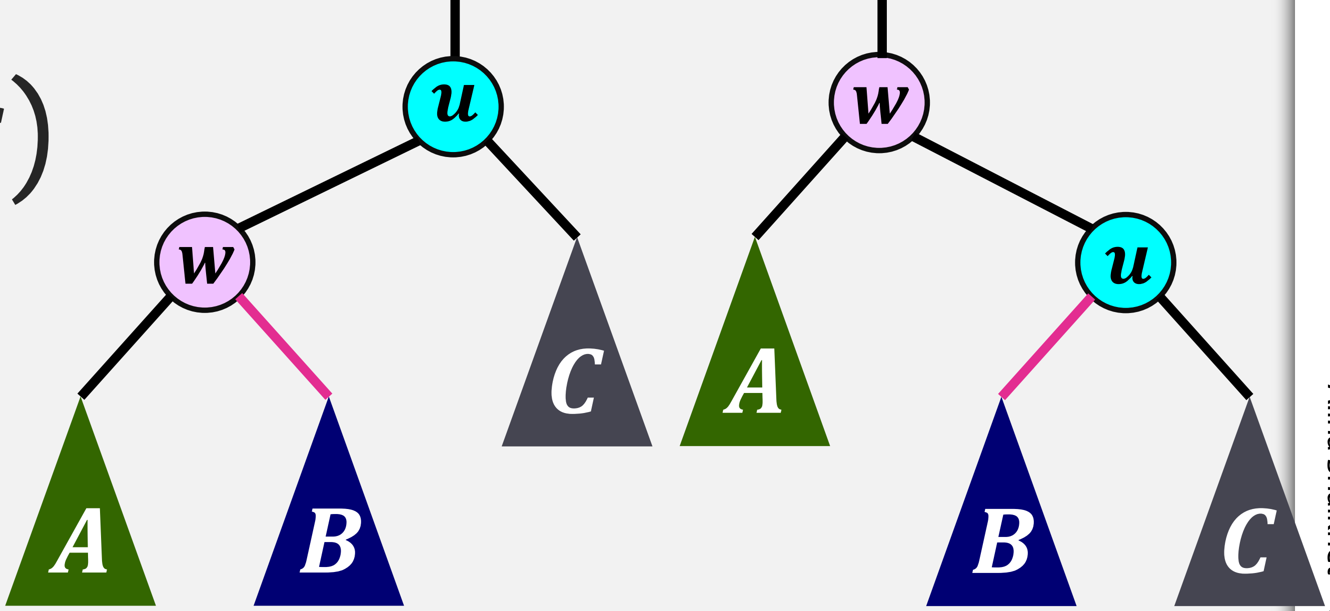
- Search for x in the tree
- If it is not there, add x as a leaf: q
- Give it a random priority: $q.p$
- As long $q.p < q.\text{parent}.p$ rotate it.



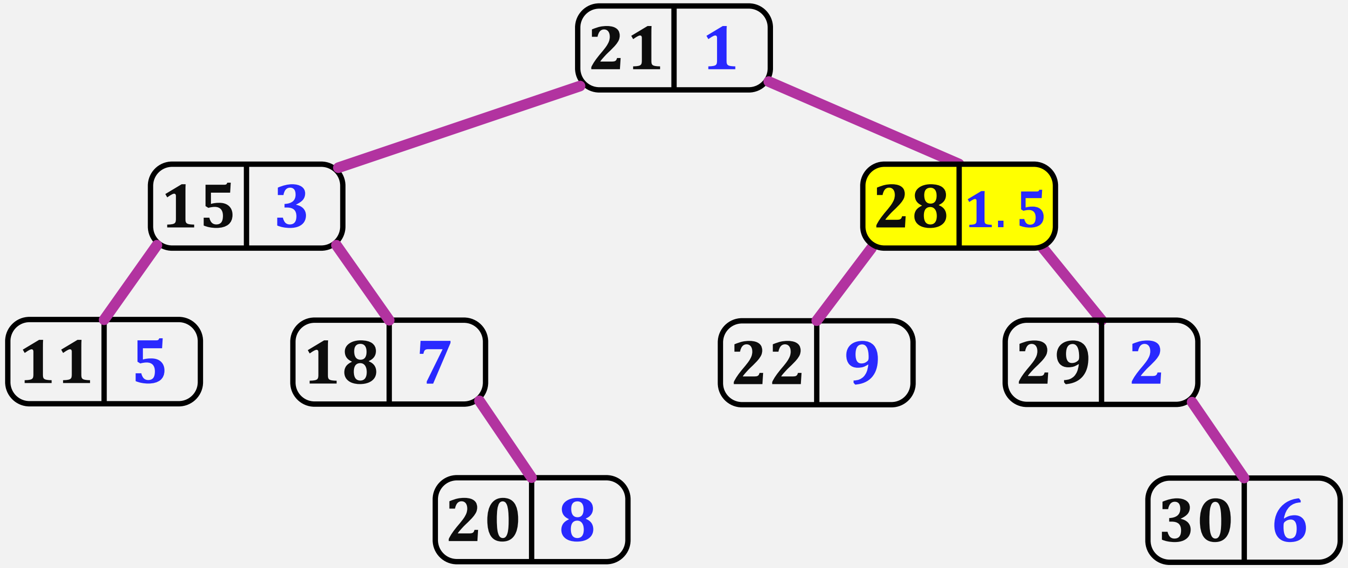


add(**28**)

Treap: SSet – add(x)



- Search for x in the tree
- If it is not there, add x as a leaf: q
- Give it a random priority: $q.p$
- As long $q.p < q.\text{parent}.p$ rotate it.



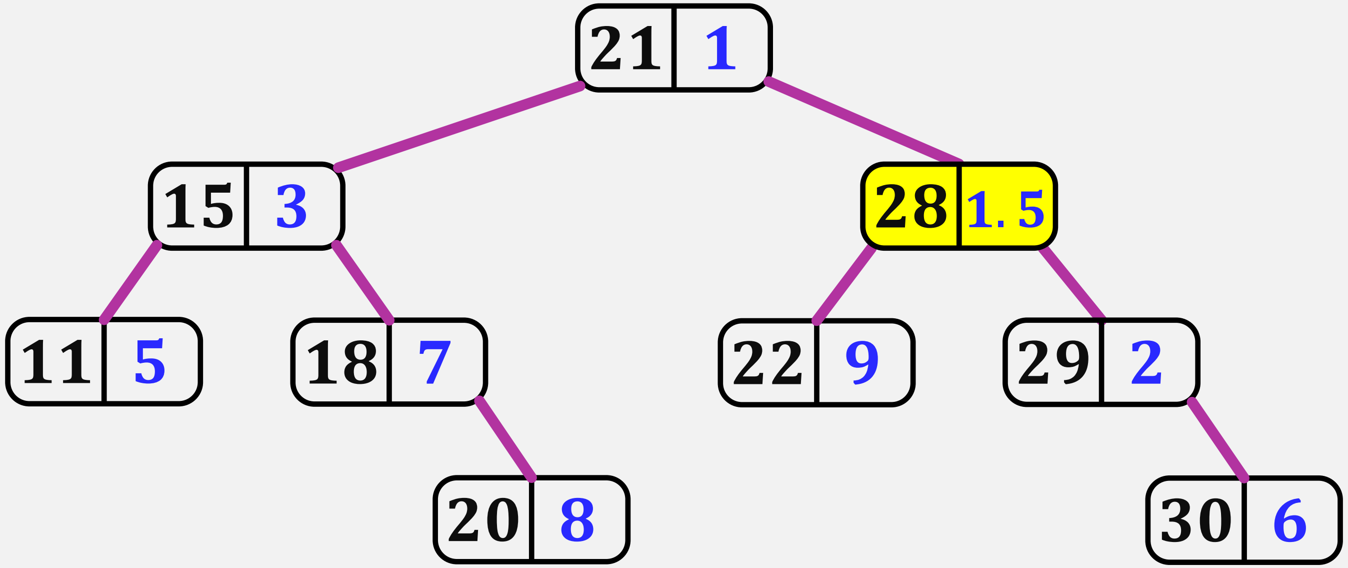
add(28)

Treap: SSet – add(x)

What is the running time of add(x) operation?

$O(\log n)$ expected

- Search for x in the tree $O(\log n)$
- If it is not there, add x as a leaf: q $O(1)$
- Give it a random priority: $q.p$ $O(1)$
- As long $q.p < q.\text{parent}.p$ rotate it. $O(\log n)$

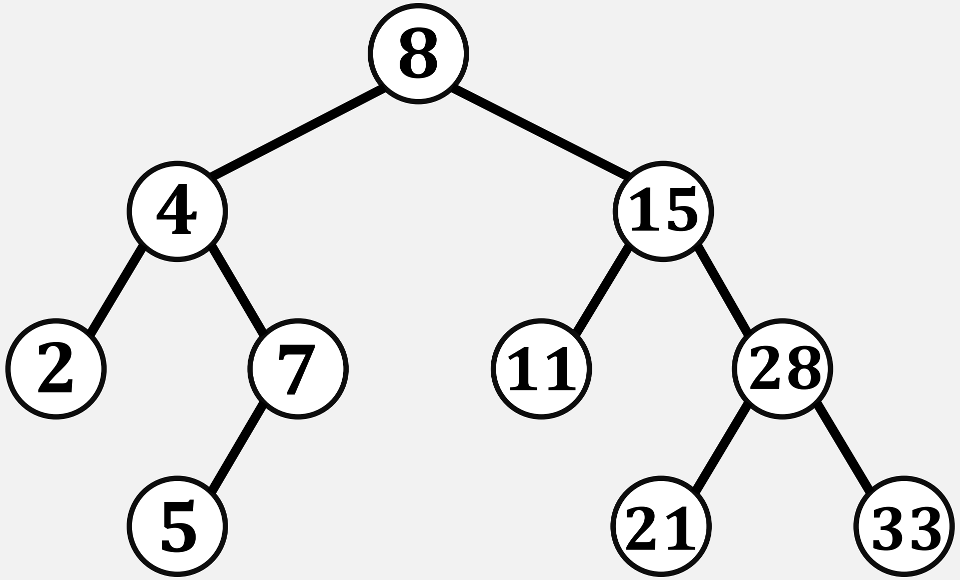


Each time we rotate we get closer to the root. That means that we can't rotate more times than the length of the search path.

Each rotation takes $O(1)$ time (it affects at most **6** pointers).

Recall – Binary Search Trees – remove($\boldsymbol{x}$)

remove($\mathbf{2}$)

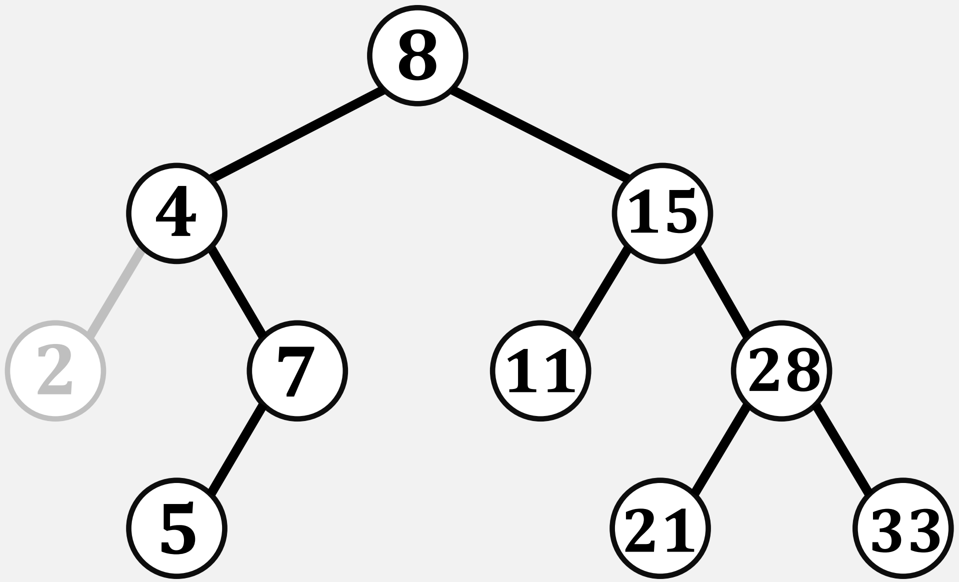


To delete a value stored in **BinarySearchTree:**

- Find node $\boldsymbol{u}$ that contains value $\boldsymbol{x}$.
- If $\boldsymbol{u}$ is a leaf, then we can just detach $\boldsymbol{u}$ from its parent.

Recall – Binary Search Trees – remove($\boldsymbol{x}$)

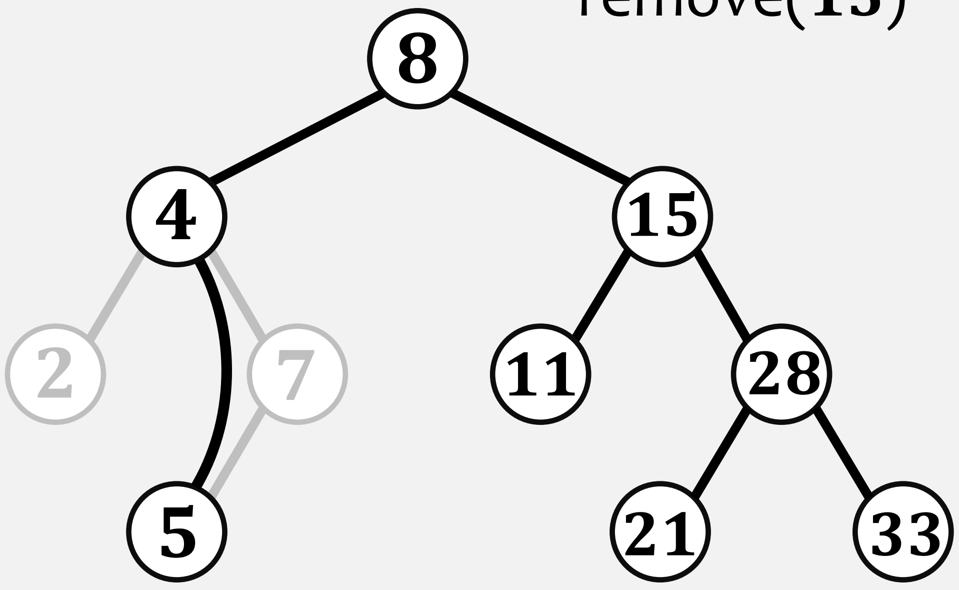
remove($\mathbf{2}$)
remove($\mathbf{7}$)



To delete a value stored in **BinarySearchTree:**

- Find node $\boldsymbol{u}$ that contains value $\boldsymbol{x}$.
- If $\boldsymbol{u}$ is a leaf, then we can just detach $\boldsymbol{u}$ from its parent.
- If $\boldsymbol{u}$ has only one child, then we can splice $\boldsymbol{u}$ from the tree by having $\boldsymbol{u}$.parent adopt $\boldsymbol{u}$'s child.

Recall – Binary Search Trees – remove($\boldsymbol{x}$)



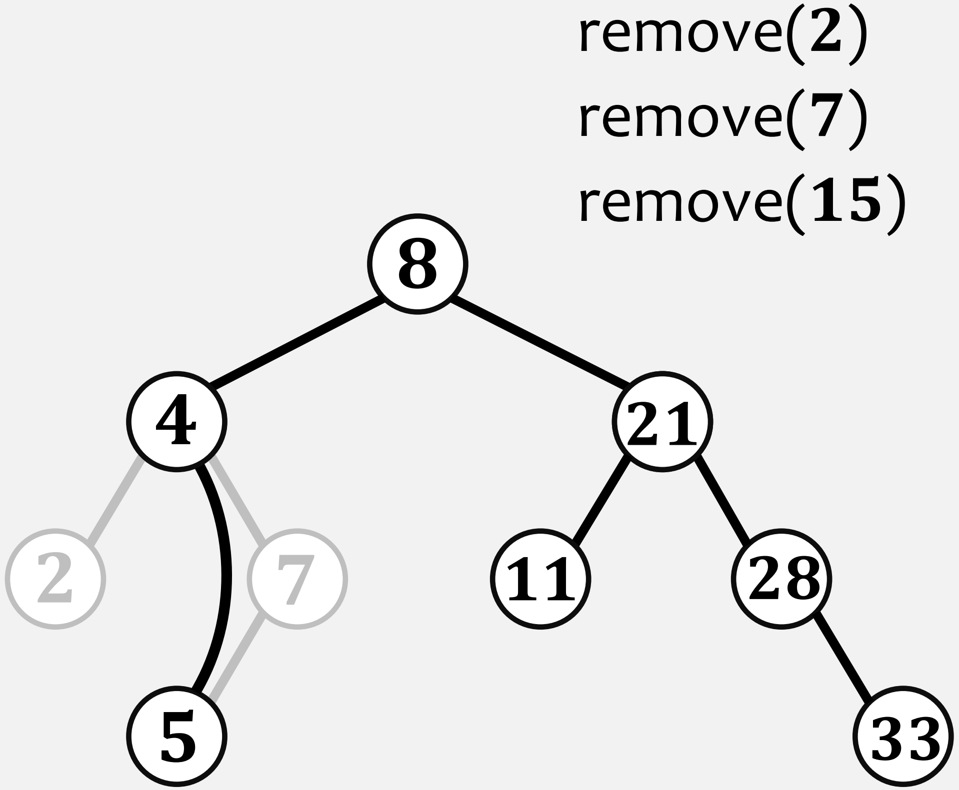
remove($\mathbf{2}$)
remove($\mathbf{7}$)
remove($\mathbf{15}$)

To delete a value stored in **BinarySearchTree:**

- Find node $\boldsymbol{u}$ that contains value $\boldsymbol{x}$.
- If $\boldsymbol{u}$ is a leaf, then we can just detach $\boldsymbol{u}$ from its parent.
- If $\boldsymbol{u}$ has only one child, then we can splice $\boldsymbol{u}$ from the tree by having $\boldsymbol{u}$.parent adopt $\boldsymbol{u}$'s child.
- If $\boldsymbol{u}$ has two children, then find a node $\boldsymbol{w}$, that has less than two children, such that $\boldsymbol{w.x}$ can replace $\boldsymbol{u.x}$.

Choose $\boldsymbol{w}$, such that $\boldsymbol{w.x}$ is the smallest value in the subtree rooted at $\boldsymbol{u}$.right. This node has no left child.

Recall – Binary Search Trees – remove($\boldsymbol{x}$)



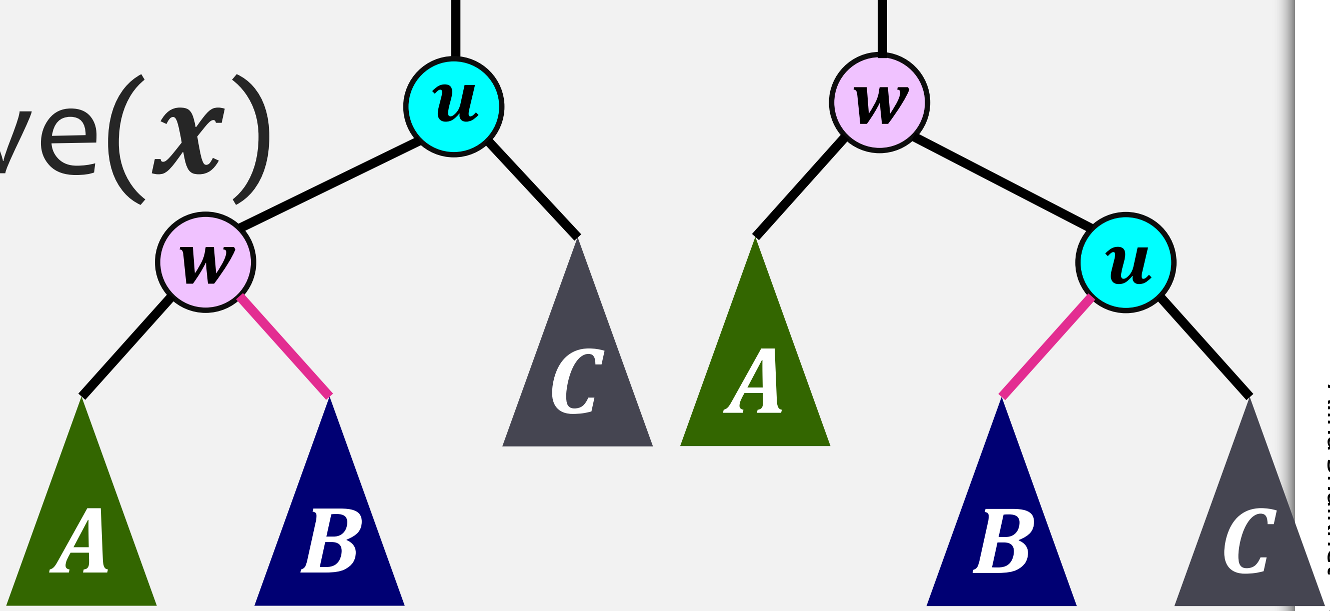
To delete a value stored in **BinarySearchTree:**

- Find node $\boldsymbol{u}$ that contains value $\boldsymbol{x}$.
- If $\boldsymbol{u}$ is a leaf, then we can just detach $\boldsymbol{u}$ from its parent.
- If $\boldsymbol{u}$ has only one child, then we can splice $\boldsymbol{u}$ from the tree by having $\boldsymbol{u}$.parent adopt $\boldsymbol{u}$'s child.
- If $\boldsymbol{u}$ has two children, then find a node $\boldsymbol{w}$, that has less than two children, such that $\boldsymbol{w.x}$ can replace $\boldsymbol{u.x}$.

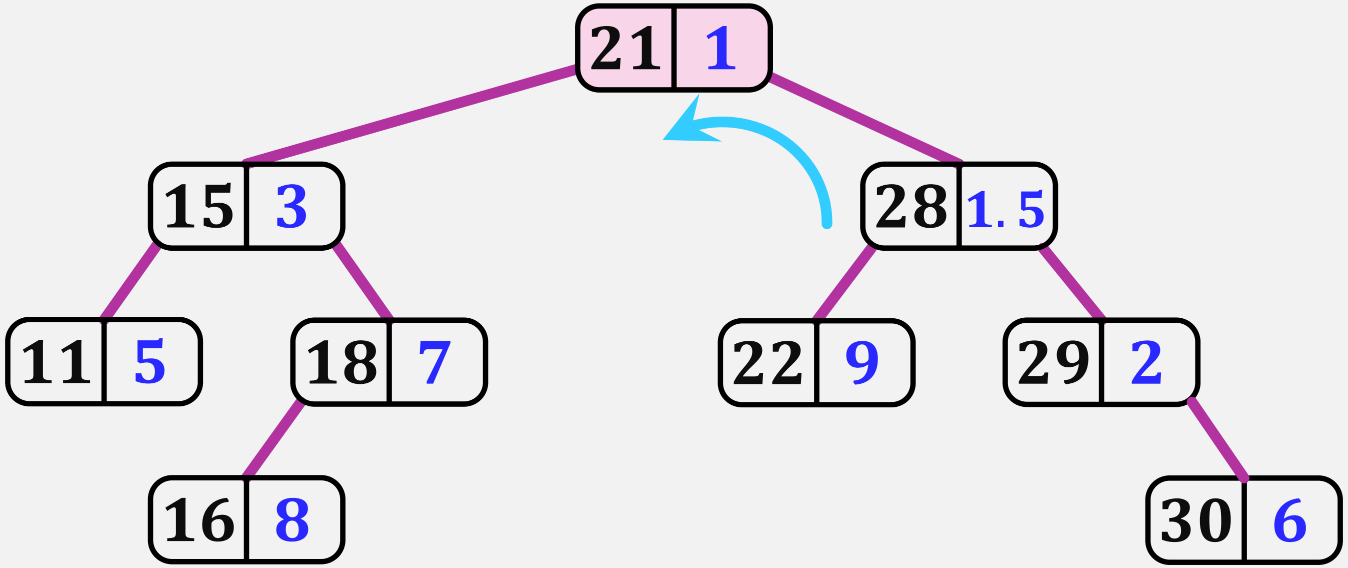
Choose $\boldsymbol{w}$, such that $\boldsymbol{w.x}$ is the smallest value in the subtree rooted at $\boldsymbol{u}$.right. This node has no left child.

Alina Shaikhet – COMP 2402 – Carleton University

Treap: SSet – remove(x)



- Find node q that contains value x.
- If q is a leaf, then we can just detach q from its parent.
- If q is not a leaf, then rotate it down until it is a leaf; then detach.



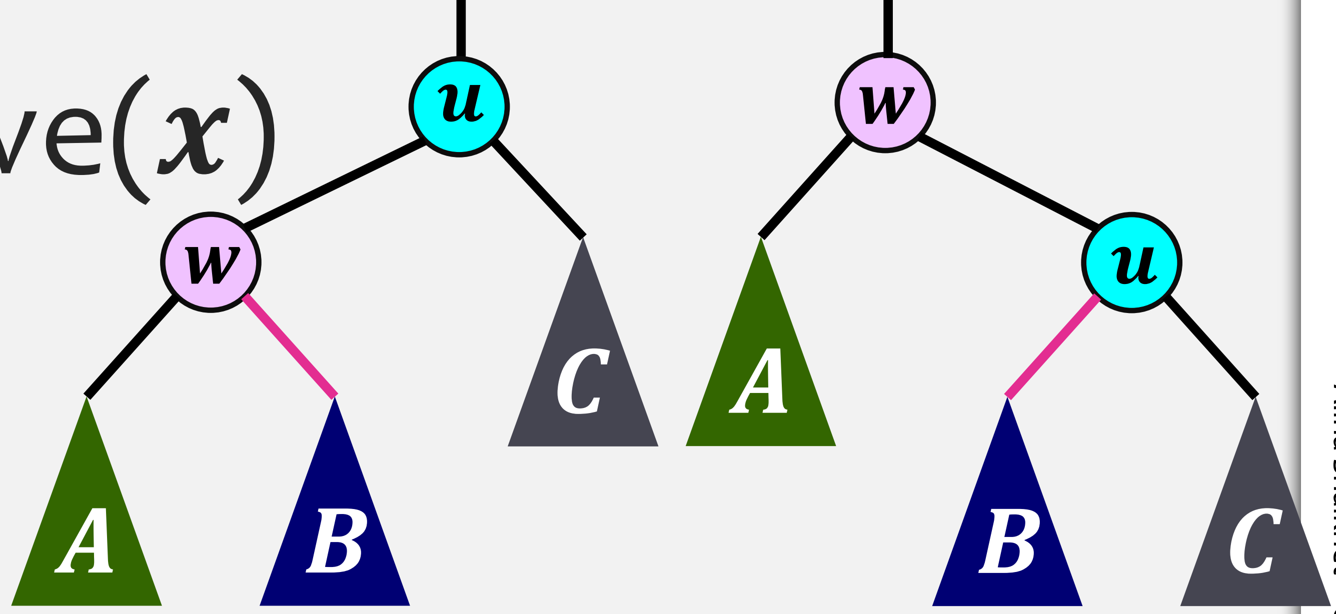
remove(21)

$$q.\text{left}.p > q.\text{right}.p$$

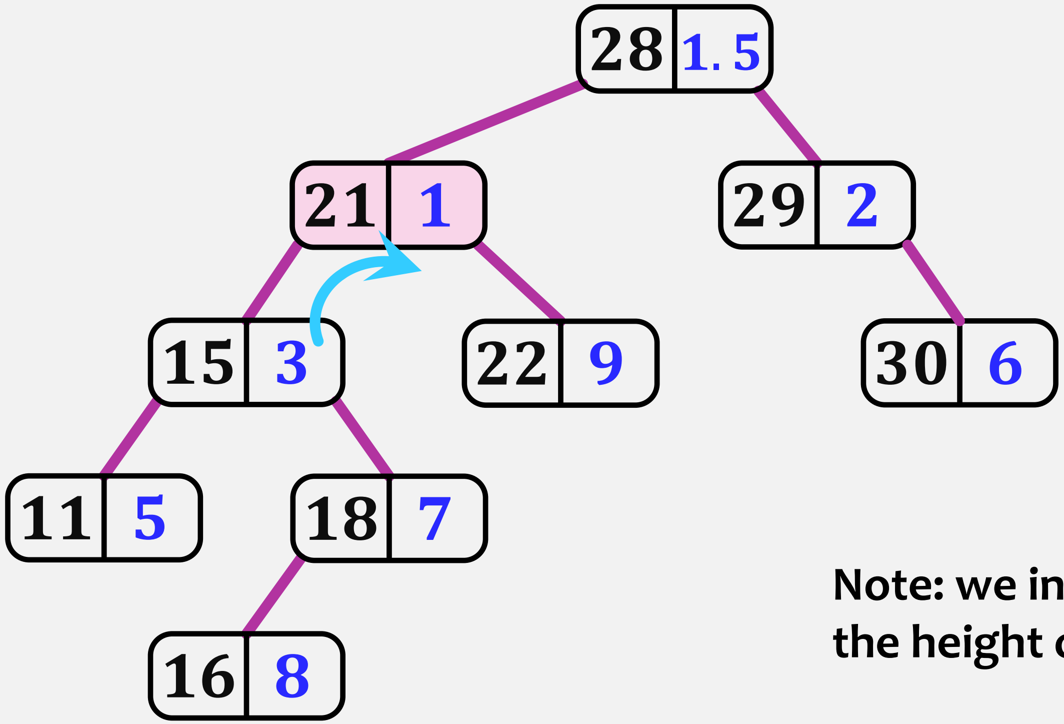
$$3 > 1.5$$

do **Left** Rotation

Treap: SSet – remove(x)



- Find node q that contains value x.
- If q is a leaf, then we can just detach q from its parent.
- If q is not a leaf, then rotate it down until it is a leaf; then detach.



Note: we increased the height of the tree

remove(**21**)

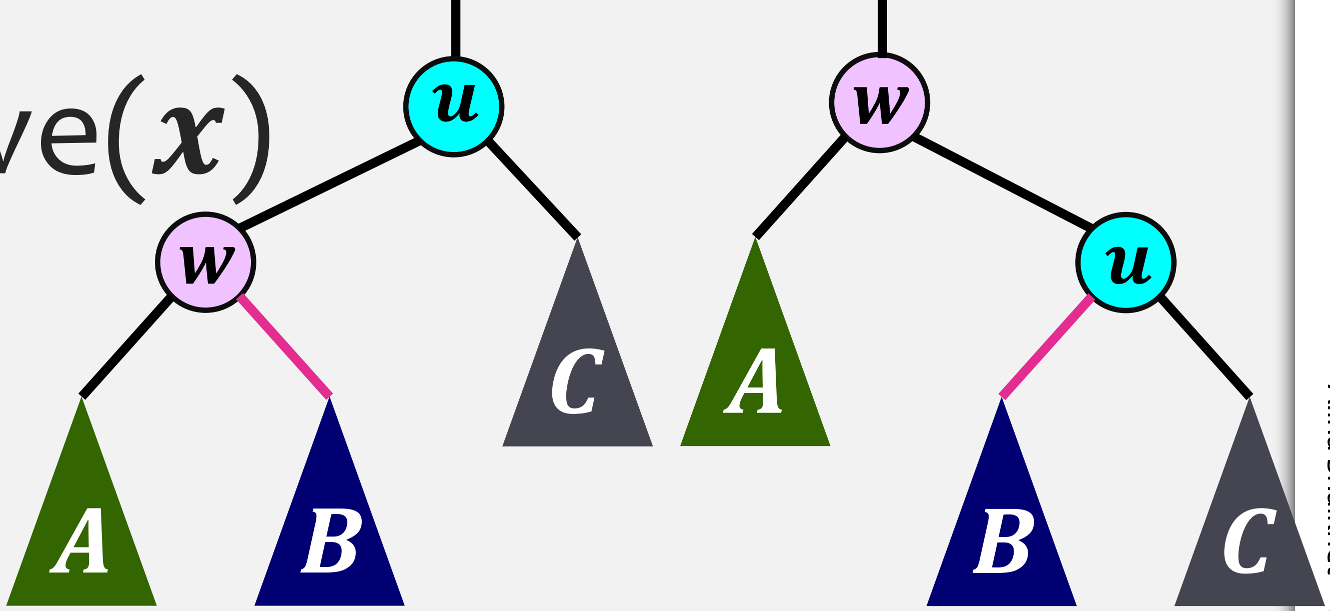
$$q.\text{left}.p < q.\text{right}.p$$

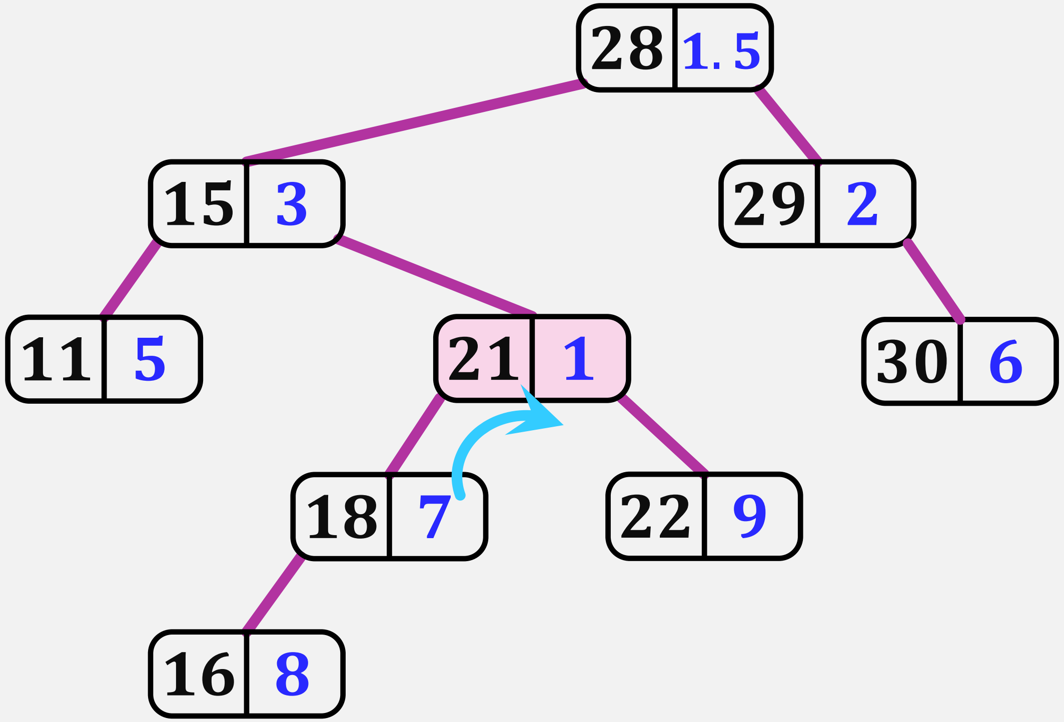
$$3 < 9$$

do **Right** Rotation

Treap: SSet – remove(x)

- Find node q that contains value x.
- If q is a leaf, then we can just detach q from its parent.
- If q is not a leaf, then rotate it down until it is a leaf; then detach.





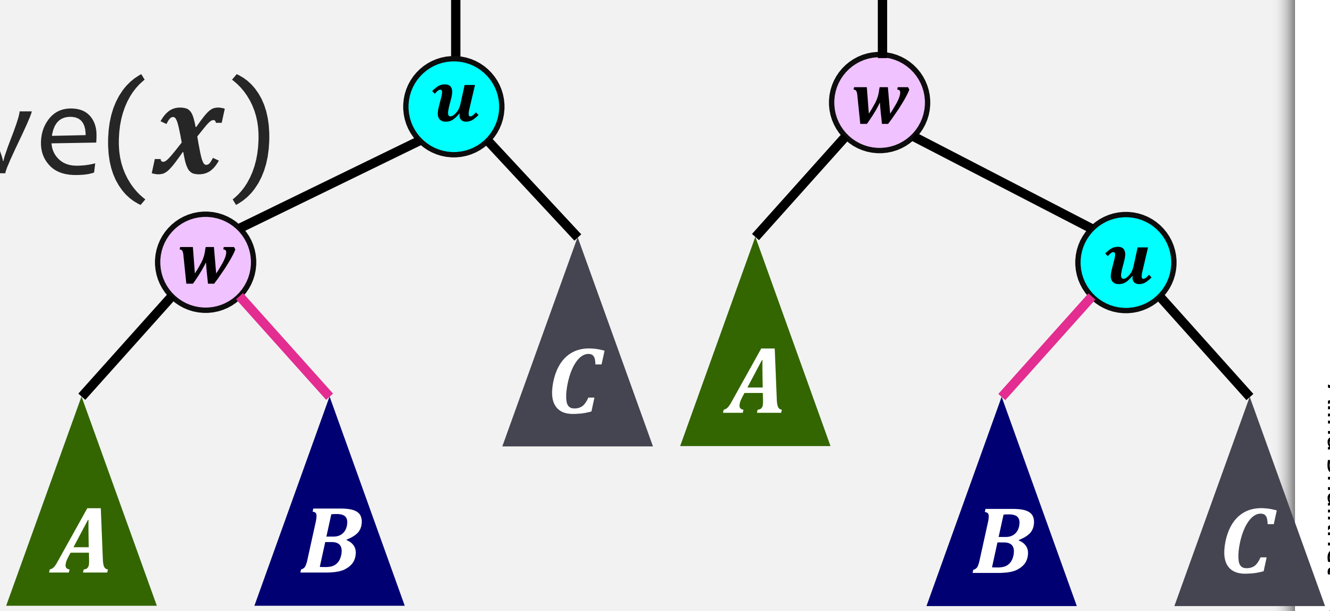
remove(**21**)

$$q.\text{left}.p < q.\text{right}.p$$

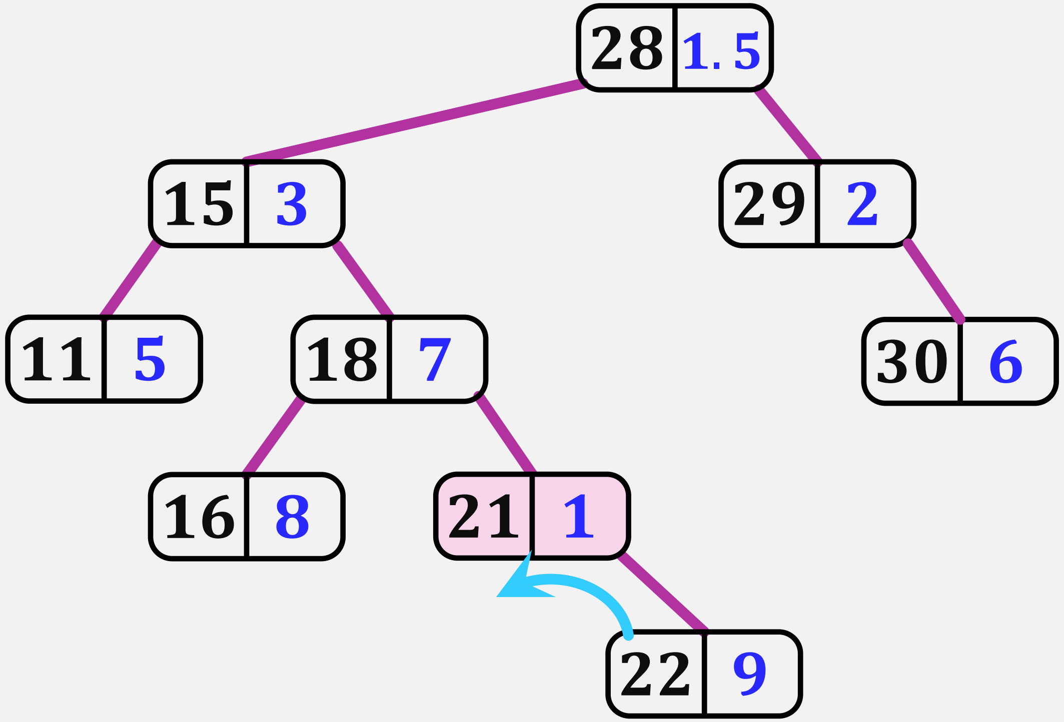
$$7 < 9$$

do **Right** Rotation

Treap: SSet – remove(x)



- Find node q that contains value x.
- If q is a leaf, then we can just detach q from its parent.
- If q is not a leaf, then rotate it down until it is a leaf; then detach.

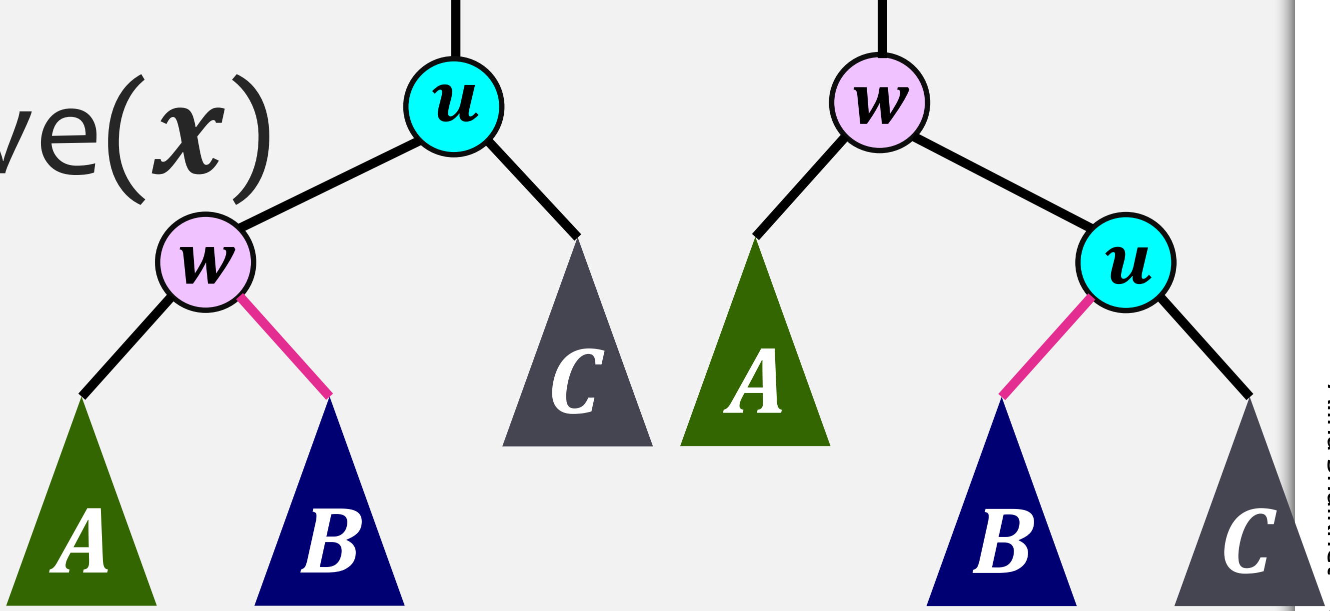


remove(**21**)

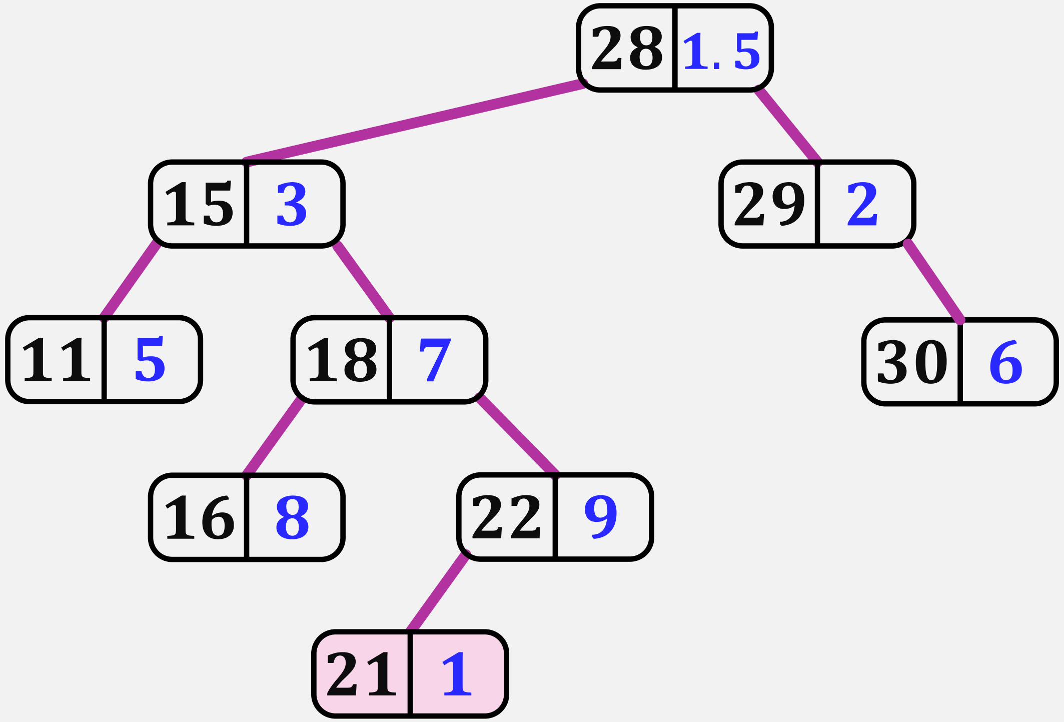
$q.\text{left} = \text{null}$

do **Left** Rotation

Treap: SSet – remove(x)



- Find node q that contains value x.
- If q is a leaf, then we can just detach q from its parent.
- If q is not a leaf, then rotate it down until it is a leaf; then detach.

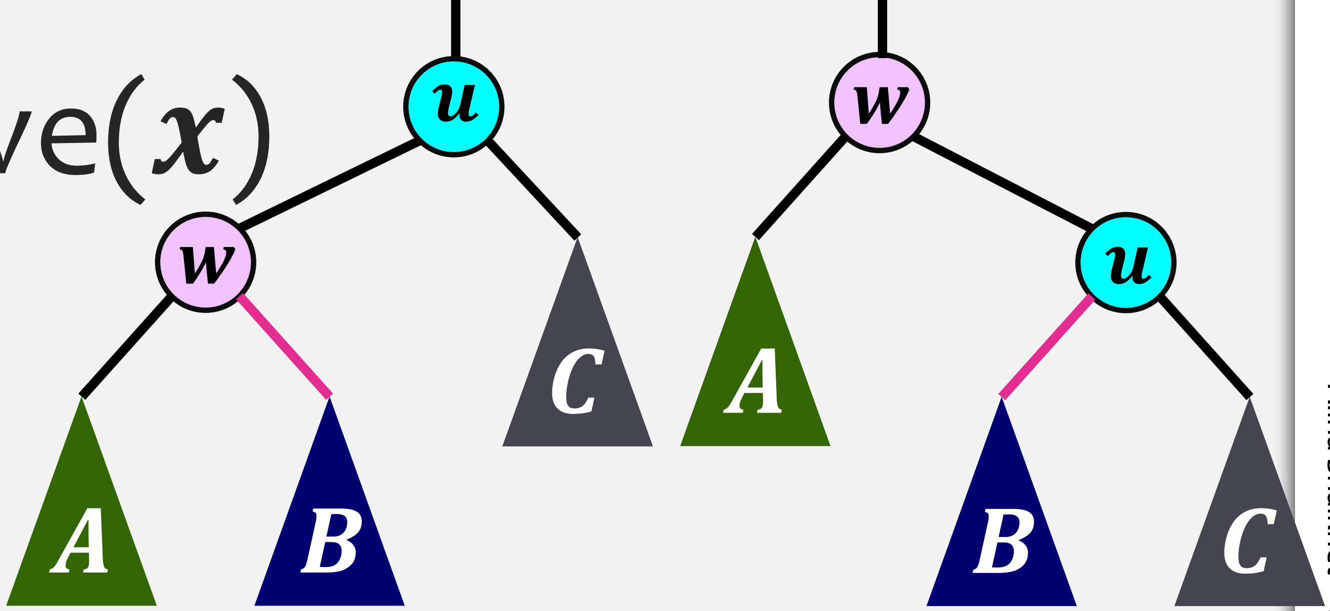


remove(**21**)

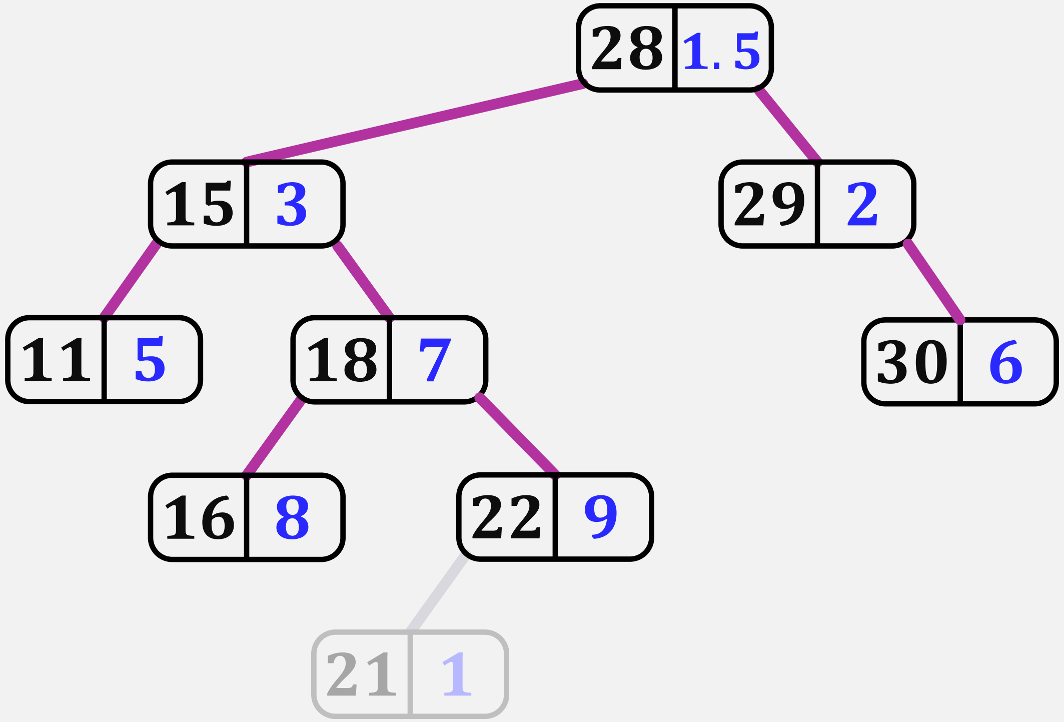
$q.\text{left} = q.\text{right} =$ null

Remove the leaf **21**

Treap: SSet – remove(x)



- Find node q that contains value x.
- If q is a leaf, then we can just detach q from its parent.
- If q is not a leaf, then rotate it down until it is a leaf; then detach.



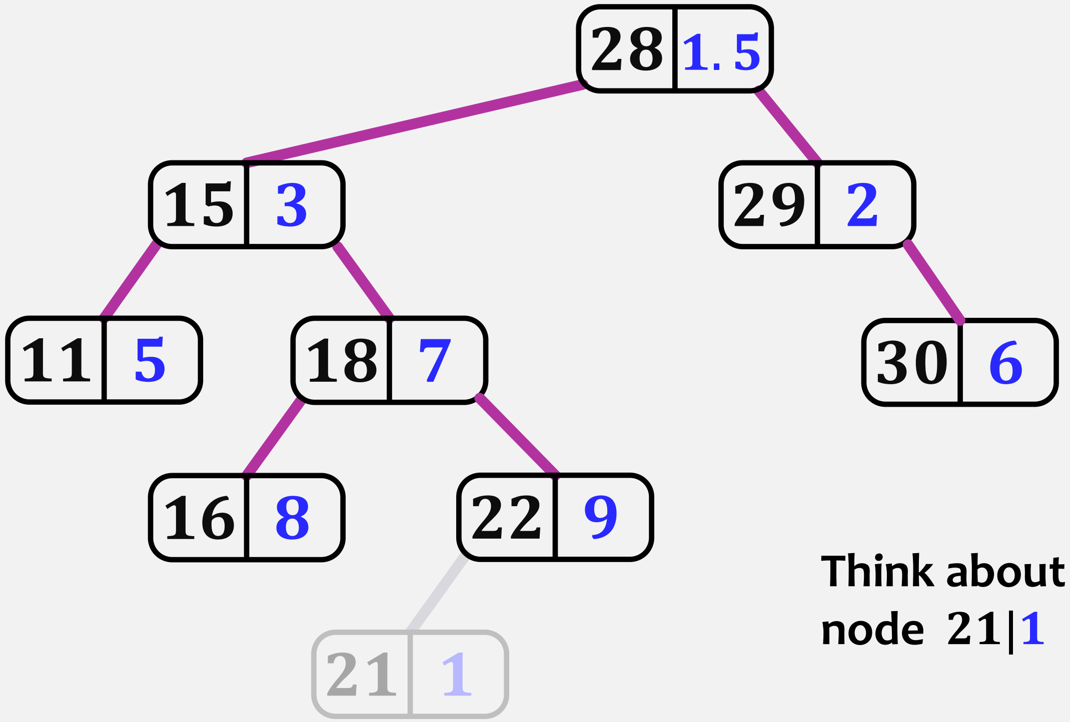
remove(21)

Treap: SSet – remove(x)

- Find node q that contains value x. $O(\log n)$
- If q is a leaf, then we can just detach q from its parent. $O(1)$
- If q is not a leaf, then rotate it down until it is a leaf; then detach. $O(\log n)$

What is the running time of remove(x) operation?

$O(\log n)$ expected



Think about adding node $21|1$

Each time we rotate we get further away from the root. That means that we can't rotate more times than the length of the longest path (which is $n-1$).

The time it takes to do a deletion on a tree with n nodes is equal to the time it takes to do an insertion on a tree with $n-1$ nodes.

Theorem 7.2

A **Treap** implements the **SSet** interface. A **Treap** supports the operations add($\boldsymbol{x}$), remove($\boldsymbol{x}$), and find($\boldsymbol{x}$) in $\boldsymbol{O}(\log \boldsymbol{n})$ expected time per operation.

The search paths in a **Treap** are considerably shorter compared to **SkipLists** and this translates into noticeably faster operations.