Carleton University

Alina Shaikhet

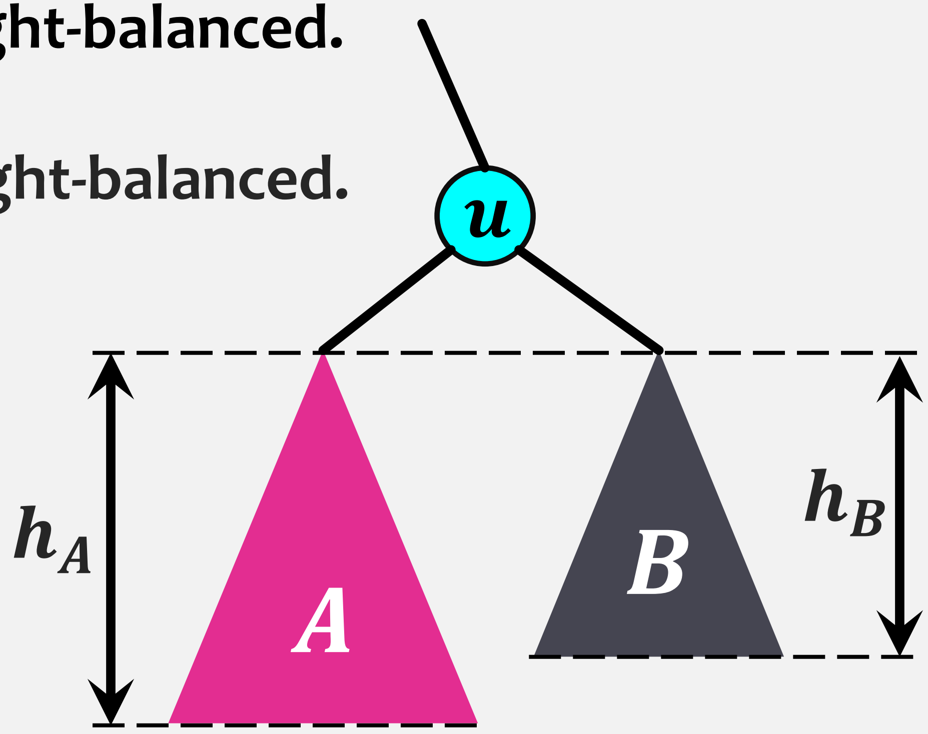
# COMP 2402
# Scapegoat Trees

# Balanced Search Trees

A node in a tree is **height-balanced** if the heights of its subtrees differ by no more than $\mathbf{1}$

If $|\boldsymbol{h_A} - \boldsymbol{h_B}| \leq \mathbf{1}$ then $\boldsymbol{u}$ is **height-balanced.**

A tree is **height-balanced** iff every node in it is **height-balanced.**

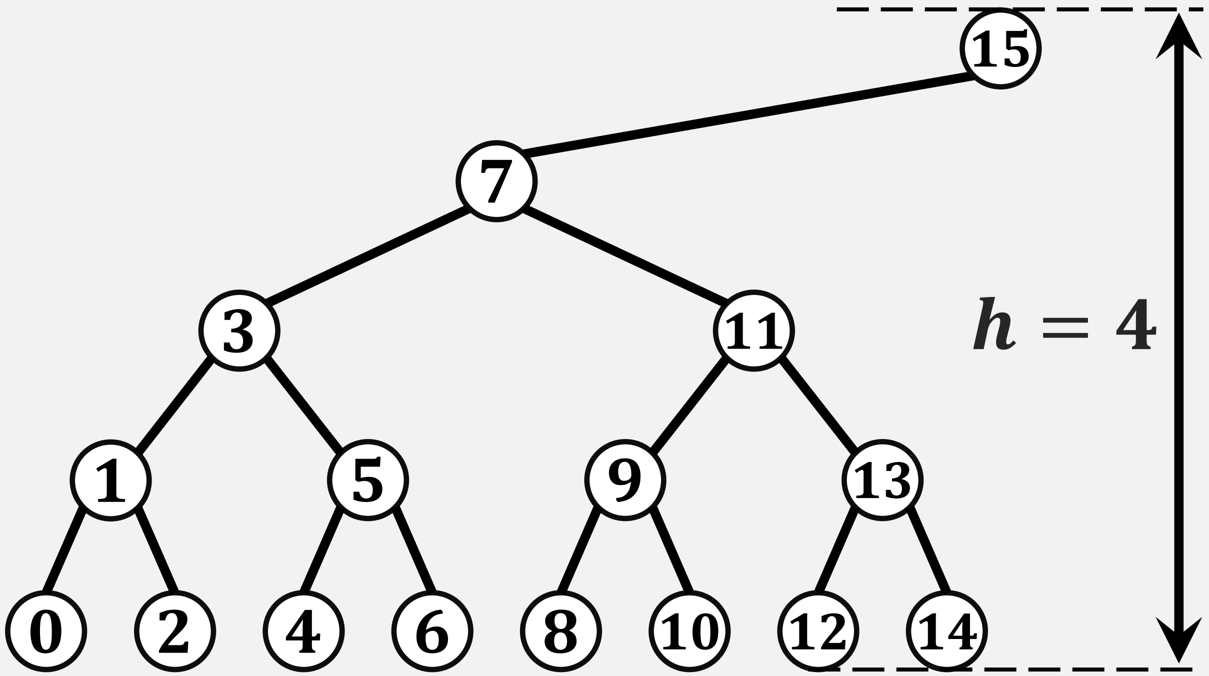



When we say that a tree is **balanced,** we usually mean that its height is $\boldsymbol{c} \log \boldsymbol{n}$, where $\boldsymbol{c}$ is some constant and $\boldsymbol{n}$ is the number of elements in the tree.

# Balanced Search Trees

Is it possible to make the height of this tree smaller?

$n = 16$




A **perfect** binary tree is a binary tree in which all interior nodes have two children, and all leaves have the same depth (or same level).

# Scapegoat Trees

A **ScapegoatTree** keeps itself balanced by performing partial rebuilding operations (where an entire subtree is deconstructed and rebuilt into a perfectly balanced subtree).

A **ScapegoatTree** is a **BinarySearchTree** that, in addition to keeping track of the number $\boldsymbol{n}$ of nodes in the tree also keeps a counter $\boldsymbol{q}$ that maintains an **upper-bound on the number of nodes**.

$$\boldsymbol{n \le q \le 2n}$$

$$\boldsymbol{q/2 \le n \le q}$$

A **ScapegoatTree** can look surprisingly unbalanced, however, it always maintains logarithmic height.

The height of the **ScapegoatTree** does not exceed

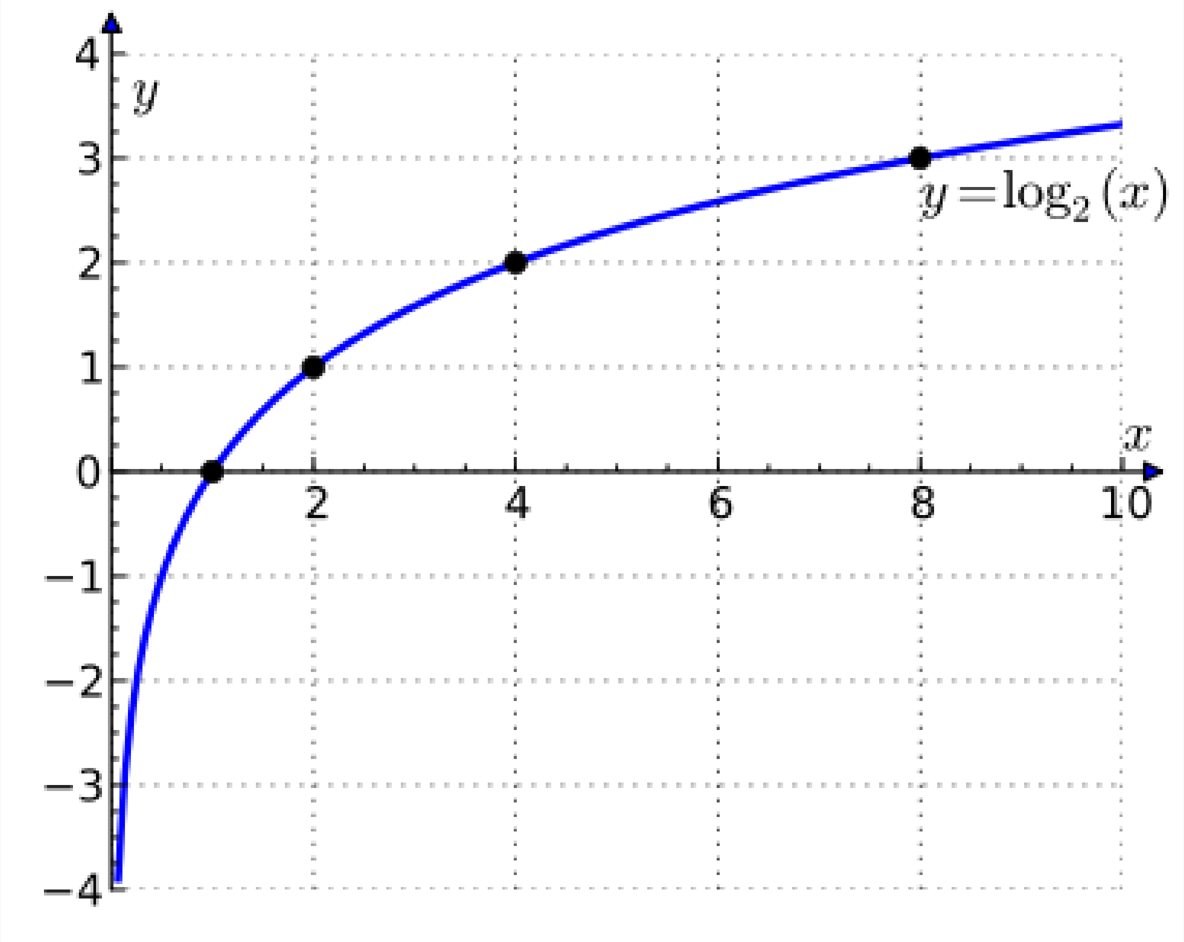
$$\log_{3/2} \boldsymbol{q} \le \log_{3/2} \boldsymbol{2n} < \log_{3/2} \boldsymbol{n} + \boldsymbol{2}$$

# Logarithms

$$y = \log_b n$$

$\log_b n$ is the number of times we divide $n$ by $b$ before it becomes $\leq 1$.

$$b^y = b^{\log_b n} = n$$




$$2^8 = 256$$
$$8 = \log_2 256$$

$$\log_b xy = \log_b x + \log_b y$$

$$\log_b n = \frac{\log_a n}{\log_a b} = \frac{\log_a n}{c} = \frac{1}{c}\log_a n$$

That is why the logarithm base is irrelevant when using Big $O$ notation.

# SSet Implementation

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

$\boldsymbol{O}(\log \boldsymbol{n})$
**ignoring rebuild()**

$\boldsymbol{O}(\log \boldsymbol{n})$

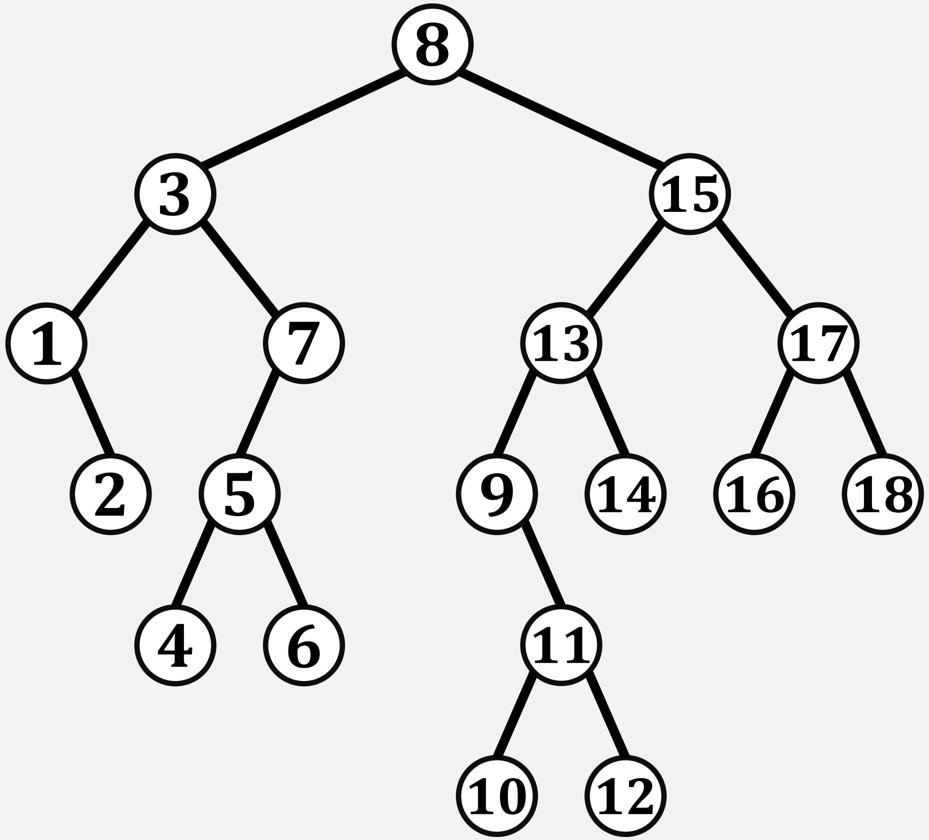
find($x$) is implemented using the standard algorithm for searching in a **BinarySearchTree.** This takes time proportional to the height of the tree which is

$$< \log_{3/2} n + 2$$

# SSet: remove($x$)

Use the usual algorithm for deleting the value $x$ stored in **BinarySearchTree:**

- Find node $u$ that contains value $x$.
- If $u$ is a leaf, then detach $u$ from its parent.
- If $u$ has only one child, then splice $u$ from the tree by having $u$.parent adopt $u$'s child.
- If $u$ has two children, then find a node $w$, that has less than two children, such that $w.x$ can replace $u.x$. Then remove $w$.
  Option **1**: Find the largest element in the left subtree of $u$,
  Option **2**: Find the smallest element in the right subtree of $u$.



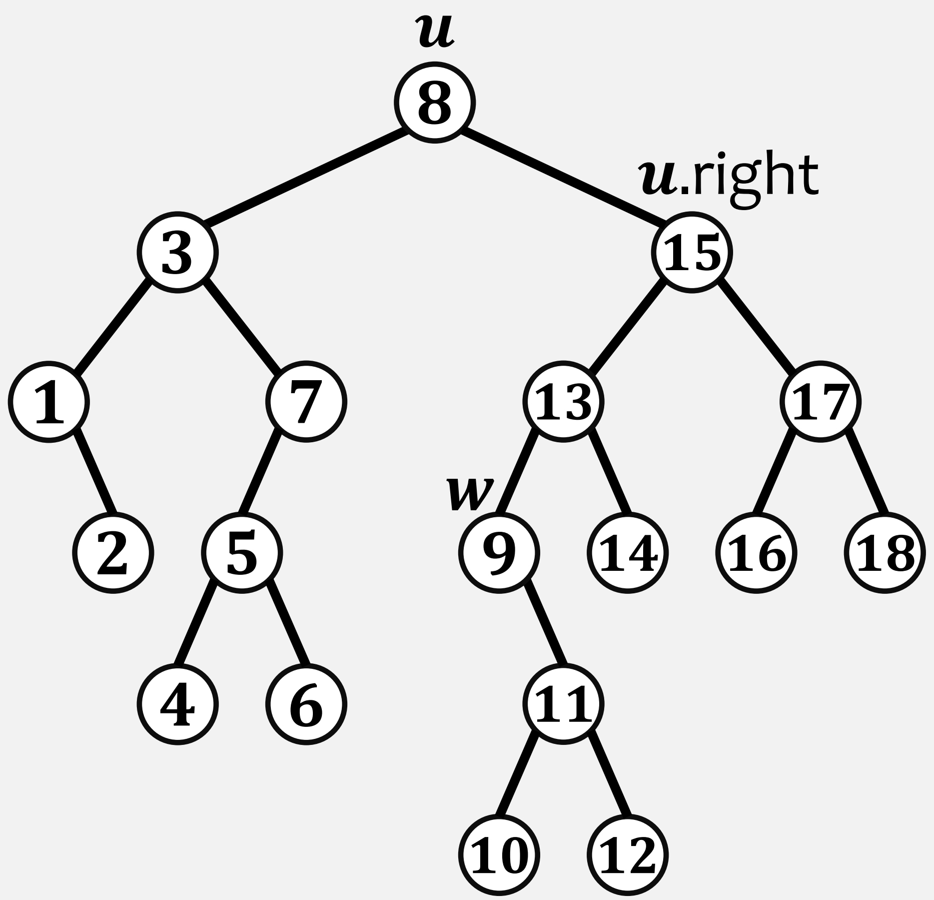
# SSet: remove($\boldsymbol{x}$)

Let's choose option $\mathbf{2}$.

Start at $\boldsymbol{u}$.right and keep going to the left child until there is no left child. The node you stopped at is $\boldsymbol{w}$.

- $\boldsymbol{w}.\boldsymbol{x}$ is the smallest value in the subtree rooted at $\boldsymbol{u}$.right.
- $\boldsymbol{w}$ has no left child.

remove($\mathbf{8}$)



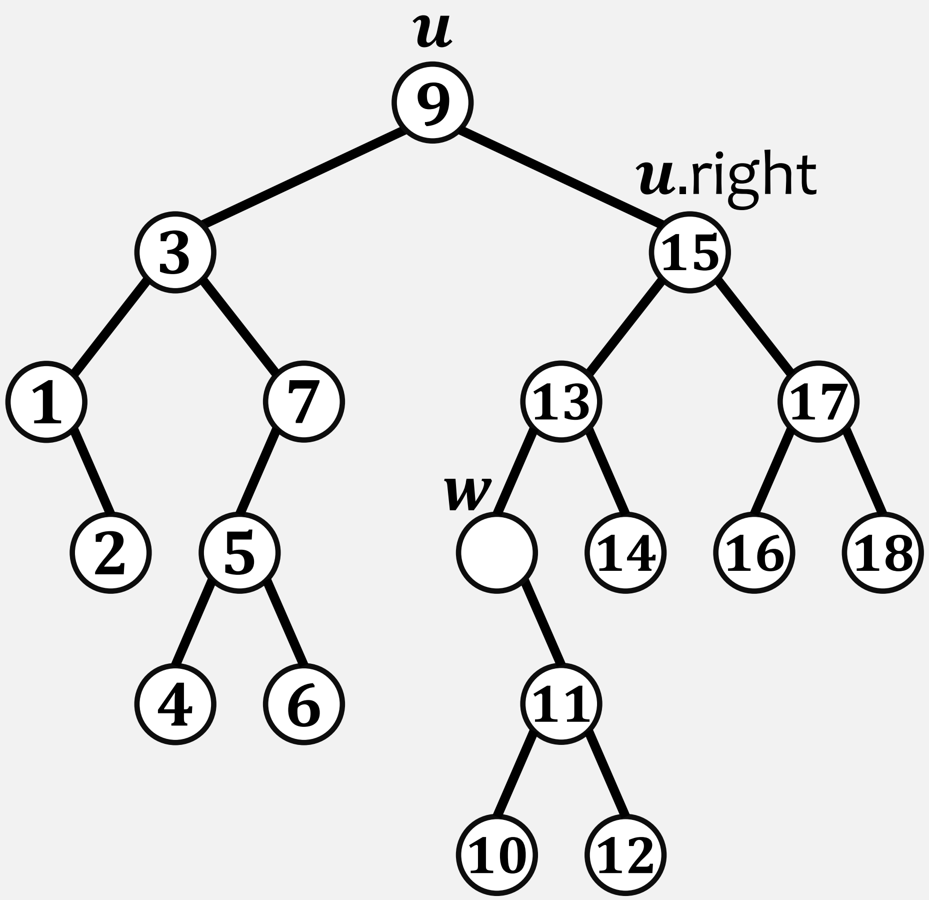
# SSet: remove($\boldsymbol{x}$)

Let's choose option $\mathbf{2}$.

Start at $\boldsymbol{u}$.right and keep going to the left child until there is no left child. The node you stopped at is $\boldsymbol{w}$.

- $\boldsymbol{w}.\boldsymbol{x}$ is the smallest value in the subtree rooted at $\boldsymbol{u}$.right.
- $\boldsymbol{w}$ has no left child.

remove($\mathbf{8}$)




delete node $\boldsymbol{w}$

# SSet: remove($\boldsymbol{x}$)

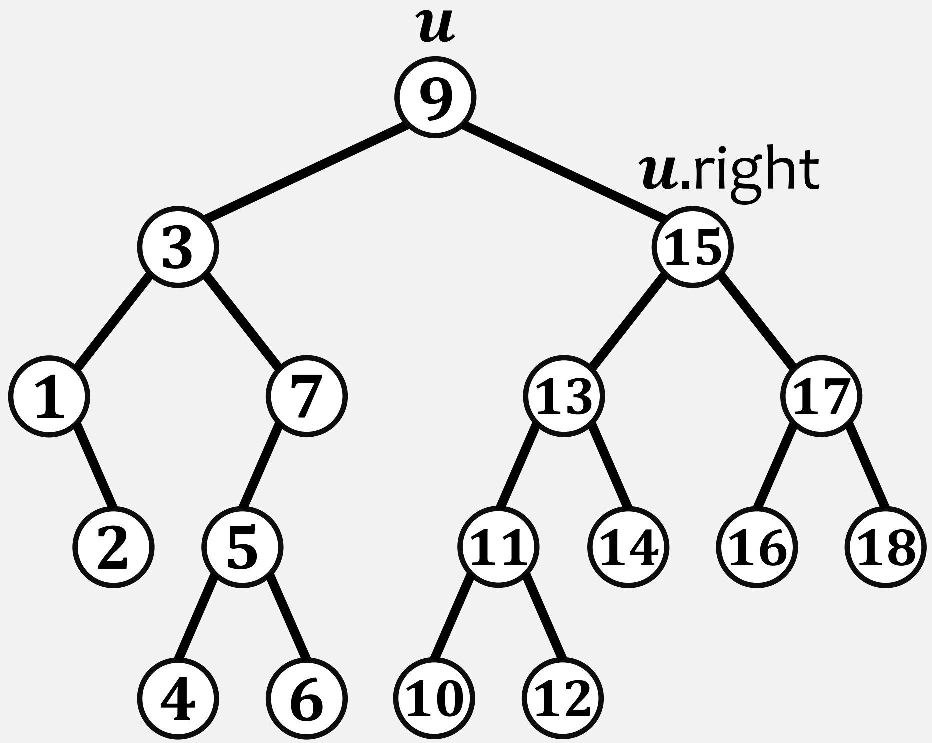
Let's choose option **2**.

Start at $\boldsymbol{u}$.right and keep going to the left child until there is no left child. The node you stopped at is $\boldsymbol{w}$.

- $\boldsymbol{w}.\boldsymbol{x}$ is the smallest value in the subtree rooted at $\boldsymbol{u}$.right.
- $\boldsymbol{w}$ has no left child.

Note that remove($\boldsymbol{x}$) can never increase the height of the tree.

remove($\mathbf{8}$)




delete node $\boldsymbol{w}$

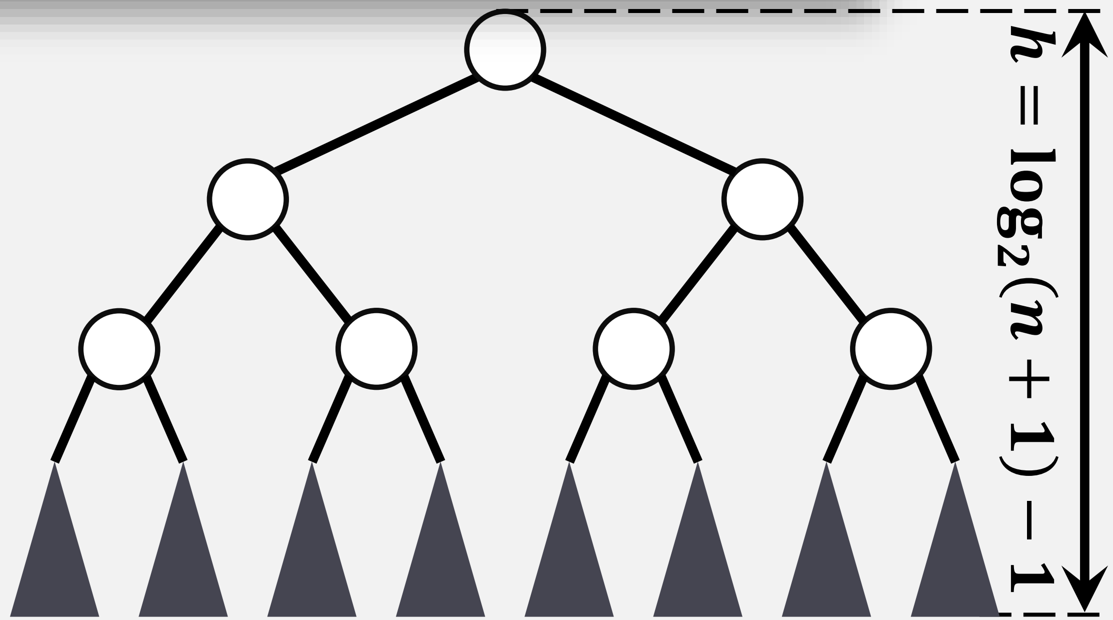
# SSet: remove($x$)

At all times: $n \leq q \leq 2n$

- remove $x$ using standard BST algorithm for removal.
- $n - -$
- if $q > 2n$ then rebuild(root) and set $q = n$

**this function rebuilds the tree into a perfectly balanced tree**

$$2^0 + 2^1 + \cdots + 2^k = 2^{k+1} - 1$$




For Complete Binary Tree

$$h = \lfloor \log_2 n \rfloor$$

There are $2^{h+1} - 1$ nodes in a **full** binary tree of height $h$:

$$2^{h+1} - 1 = n$$

$$2^{h+1} = n + 1$$

$$h + 1 = \log_2 2^{h+1} = \log_2(n+1)$$

# SSet: remove($\boldsymbol{x}$)

At all times: $\boldsymbol{n \le q \le 2n}$

- remove $\boldsymbol{x}$ using standard BST algorithm for removal.
- $\boldsymbol{n - -}$
- if $\boldsymbol{q > 2n}$ then rebuild(root) and set $\boldsymbol{q = n}$

$\boldsymbol{O(n)}$

**this function rebuilds the tree into a perfectly balanced tree**

After rebuild() for every node $\boldsymbol{u}$ :

$$|\text{size}(\boldsymbol{u}.\text{left}) - \text{size}(\boldsymbol{u}.\text{right})| \le \mathbf{1}$$

If we ignore the cost of rebuilding, the running time of the remove($\boldsymbol{x}$) operation is proportional to the height of the tree and is therefore $\boldsymbol{O(\log n)}$.

# SSet: add($x$)

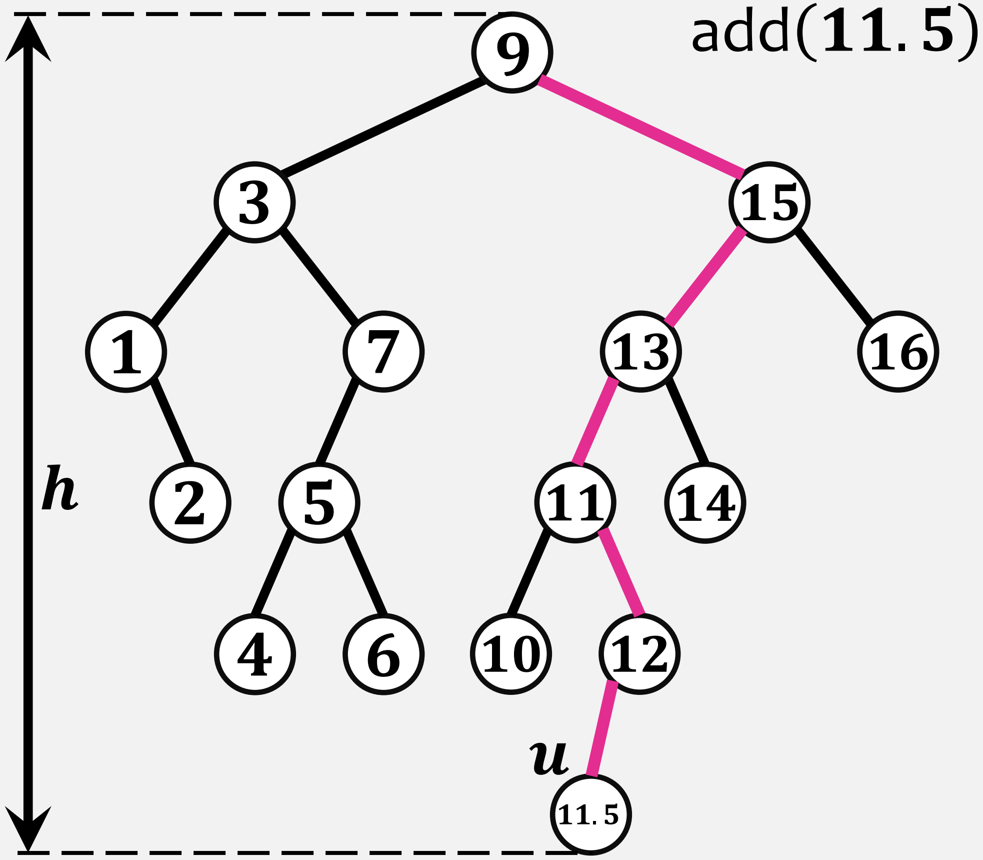
At all times: $n \le q \le 2n$
$h \le \log_{3/2} q < \log_{3/2} n + 2$

- Use the standard algorithm for adding $x$ to **BinarySearchTree**:
- $n{+}{+}$
- $q{+}{+}$
- If $(\text{depth}(u) > \log_{3/2} q)$ then walk from $u$ back up to the root looking for a scapegoat $w$:

$$\frac{\text{size}(w.\text{child})}{\text{size}(w)} > \frac{2}{3}$$

$w$.child is a child of $w$ on the path from the root to $u$

Then rebuild($w$) $O(\text{size}(w))$



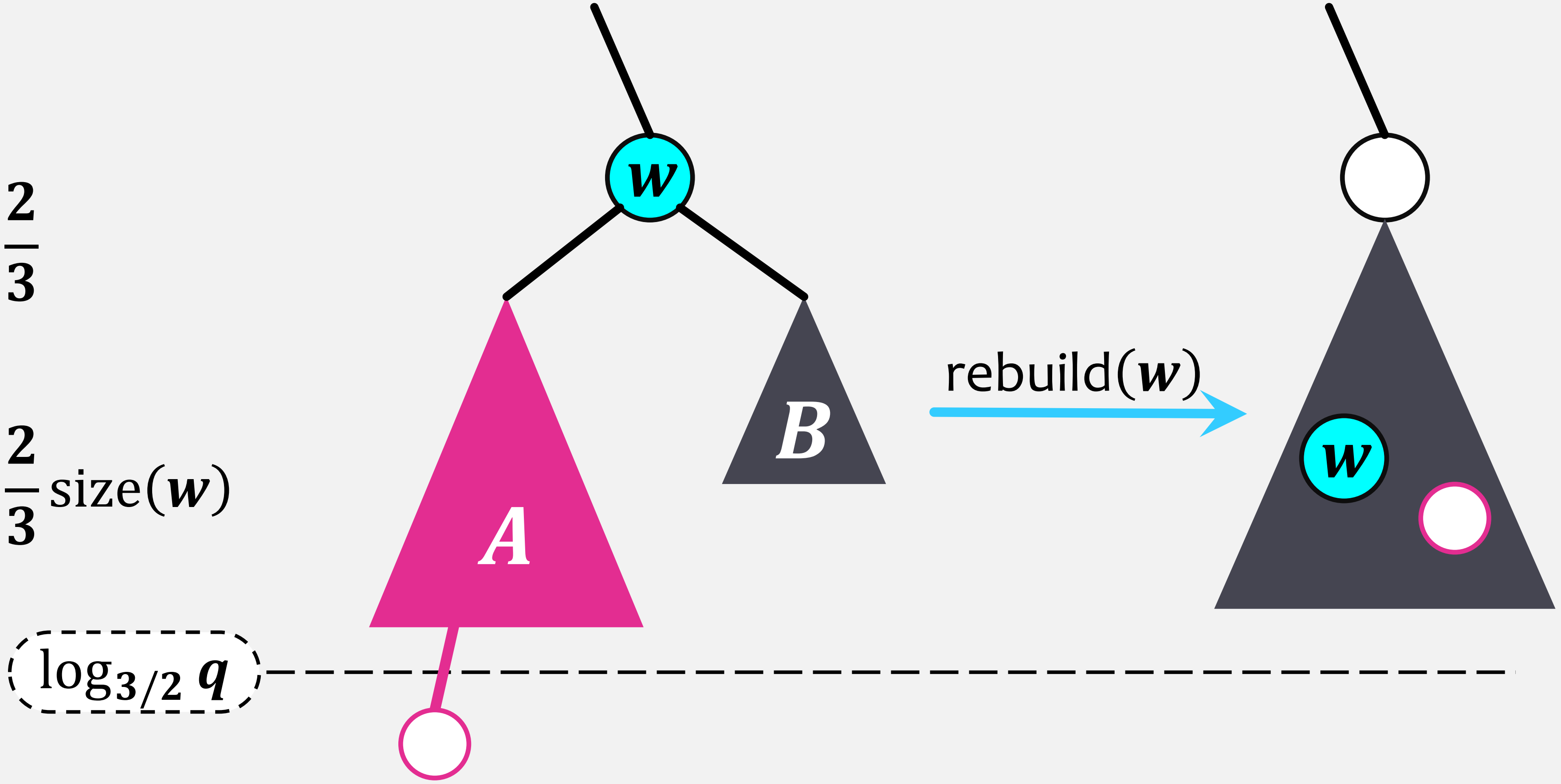

If we ignore the cost of finding the scapegoat and rebuilding: $O(\log n)$

add($x$) can increase the height of a tree $\log_{3/2} q$

# SSet: add($\boldsymbol{x}$) – scapegoat $\boldsymbol{w}$

$$\frac{\text{size}(\boldsymbol{A} + \bigcirc)}{\text{size}(\boldsymbol{w})} > \frac{\mathbf{2}}{\mathbf{3}}$$

$$\text{size}(\boldsymbol{A} + \bigcirc) > \frac{\mathbf{2}}{\mathbf{3}}\,\text{size}(\boldsymbol{w})$$



# Does the scapegoat exist?

At all times: $\boldsymbol{n} \leq \boldsymbol{q} \leq \boldsymbol{2n}$

Suppose, that there does not exist a scapegoat $\boldsymbol{w}$ on the path from $\boldsymbol{u}$ (with value $\boldsymbol{x}$) to the root.

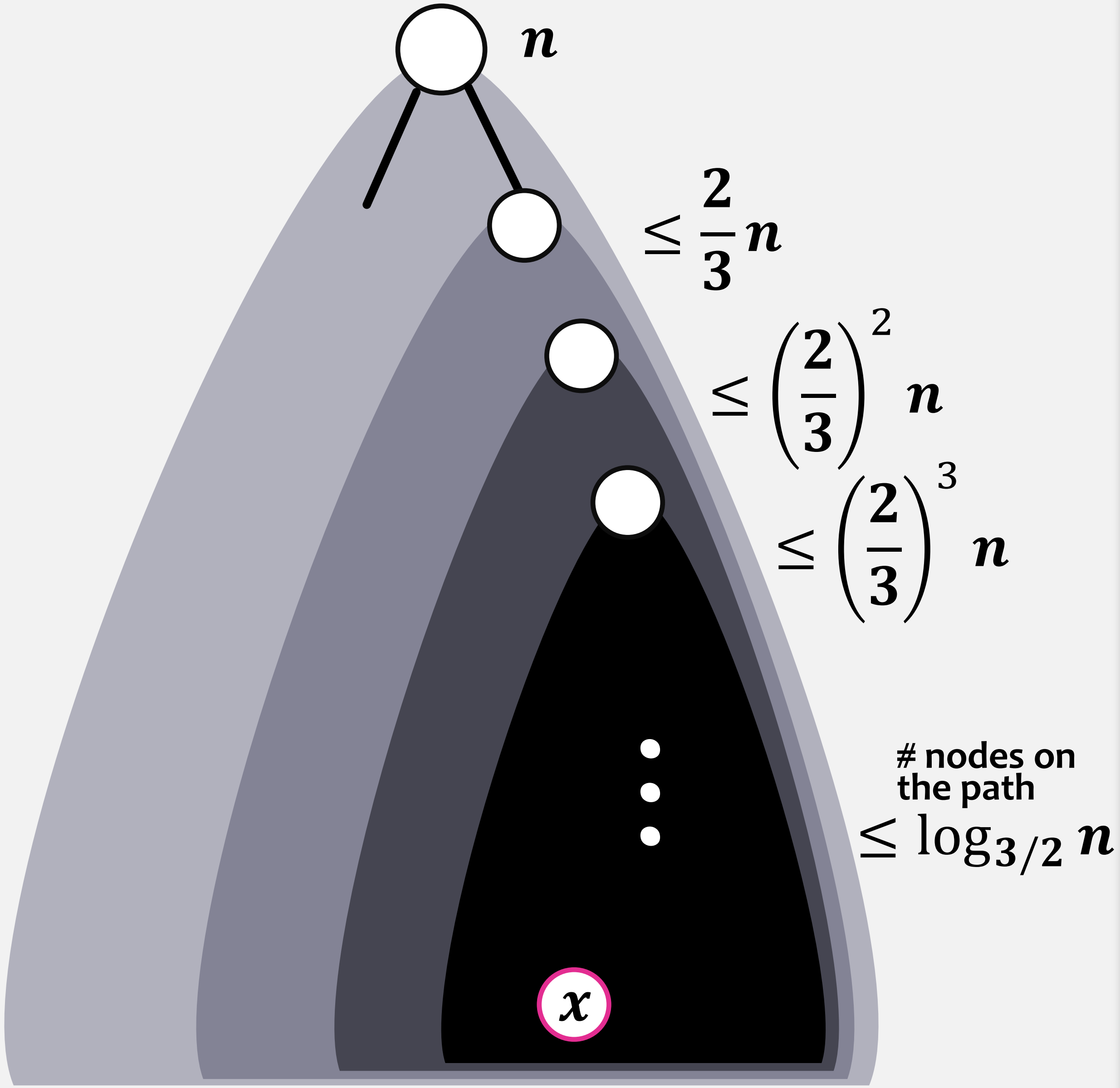
Then for all the nodes $\boldsymbol{w}$ on the path

$$\frac{\text{size}(\boldsymbol{w})}{\text{size}(\text{parent of } \boldsymbol{w})} \leq \frac{\mathbf{2}}{\mathbf{3}}$$

Thus, the length of the path from the root to $\boldsymbol{u}$ is at most $\log_{3/2} \boldsymbol{n} \leq \log_{3/2} \boldsymbol{q}$

So, we didn't cross the $\log_{3/2} \boldsymbol{q}$ line.

Contradiction!



# Scapegoat Trees

A **ScapegoatTree** can look surprisingly unbalanced, however, it always maintains logarithmic height:

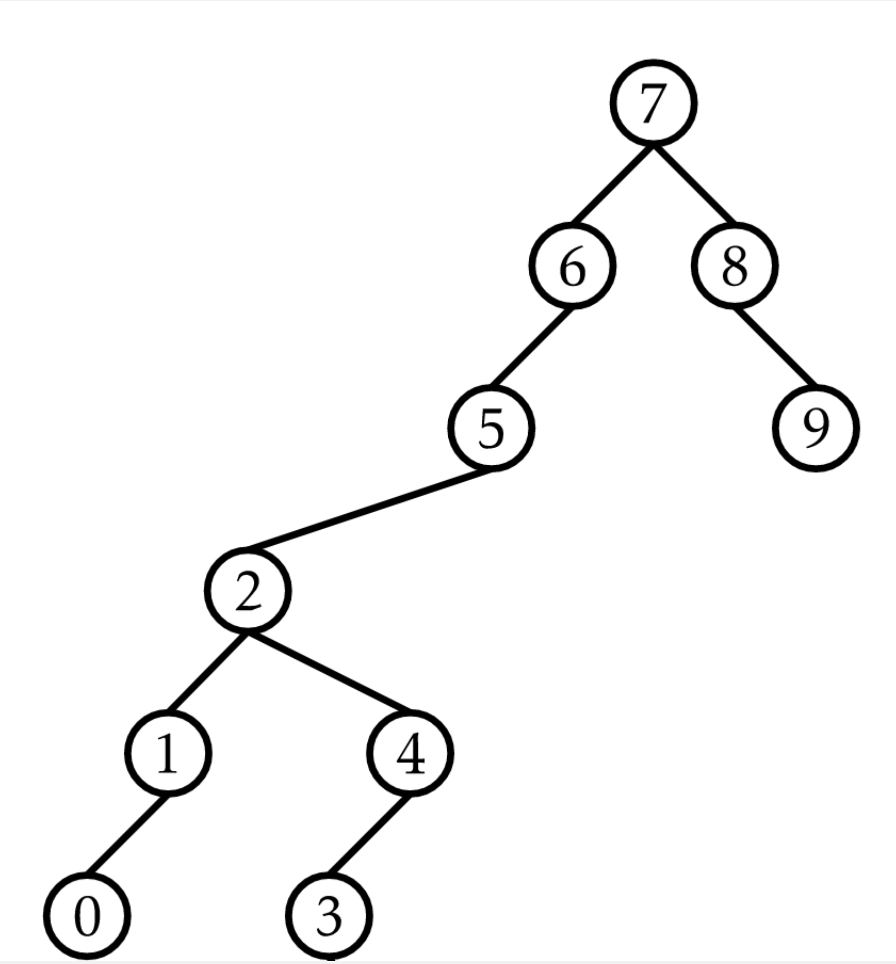
$$\boldsymbol{h} \leq \log_{3/2} \boldsymbol{q}$$

For the tree in the example:

$$\boldsymbol{n} = \mathbf{10} = \boldsymbol{q}$$

$$\text{height} = \mathbf{5}$$

$$\mathbf{5} < \log_{3/2} \mathbf{10} \approx \mathbf{5.679}$$




from the ods textbook

# Scapegoat Trees

A **ScapegoatTree** can look surprisingly unbalanced, however, it always maintains logarithmic height:

$$h \leq \log_{3/2} q$$

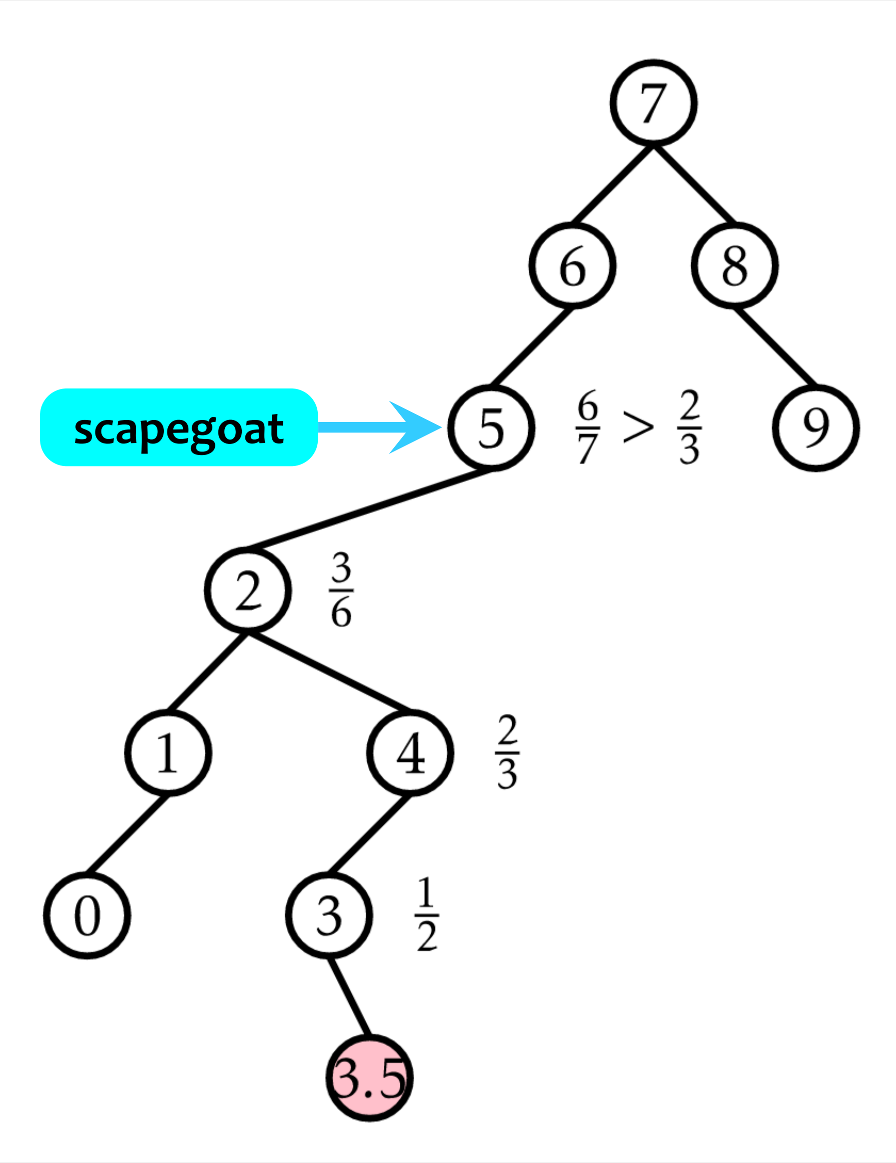
For the tree in the example:

$$n = \cancel{10}\ 11 = q$$

$$\text{height} = \cancel{5}\ 6$$

$$5 < \log_{3/2} 10 \approx 5.679$$

$$6 > \log_{3/2} 11 \approx 5.914$$




from the ods textbook

# SSet: add($x$)

```
boolean add(T x) {
  // first do basic insertion keeping track of depth
  Node<T> u = newNode(x);
  int d = addWithDepth(u);
  if (d > log32(q)) {
    // depth exceeded, find scapegoat
    Node<T> w = u.parent;
    while (3*size(w) <= 2*size(w.parent))
      w = w.parent;
    rebuild(w.parent);
  }
  return d >= 0;
}
```

# rebuild($\boldsymbol{u}$)

- traverse (in-order) $\boldsymbol{u}$'s subtree and collect all its nodes into an array $\boldsymbol{a}$
- recursively build a balanced subtree using $\boldsymbol{a}$:

$$\boldsymbol{m} = \frac{\boldsymbol{a}.\text{length}}{\boldsymbol{2}}$$

$\boldsymbol{a}[\boldsymbol{m}]$ becomes the root of the new subtree,
$\boldsymbol{a}[\boldsymbol{0}], \dots, \boldsymbol{a}[\boldsymbol{m}-\boldsymbol{1}]$ get stored recursively in the **left** subtree, and
$\boldsymbol{a}[\boldsymbol{m}+\boldsymbol{1}], \dots, \boldsymbol{a}[\boldsymbol{a}.\text{length}-\boldsymbol{1}]$ get stored recursively in the right subtree.

# Analysis

**Lemma 8.3**: Starting with an empty **ScapegoatTree** over any sequence of $m$ add($x$) and remove($x$) operations the total time spent in calls to rebuild($u$) is $O(m \log N)$, where $N$ is the maximum value of $n$ at any time.

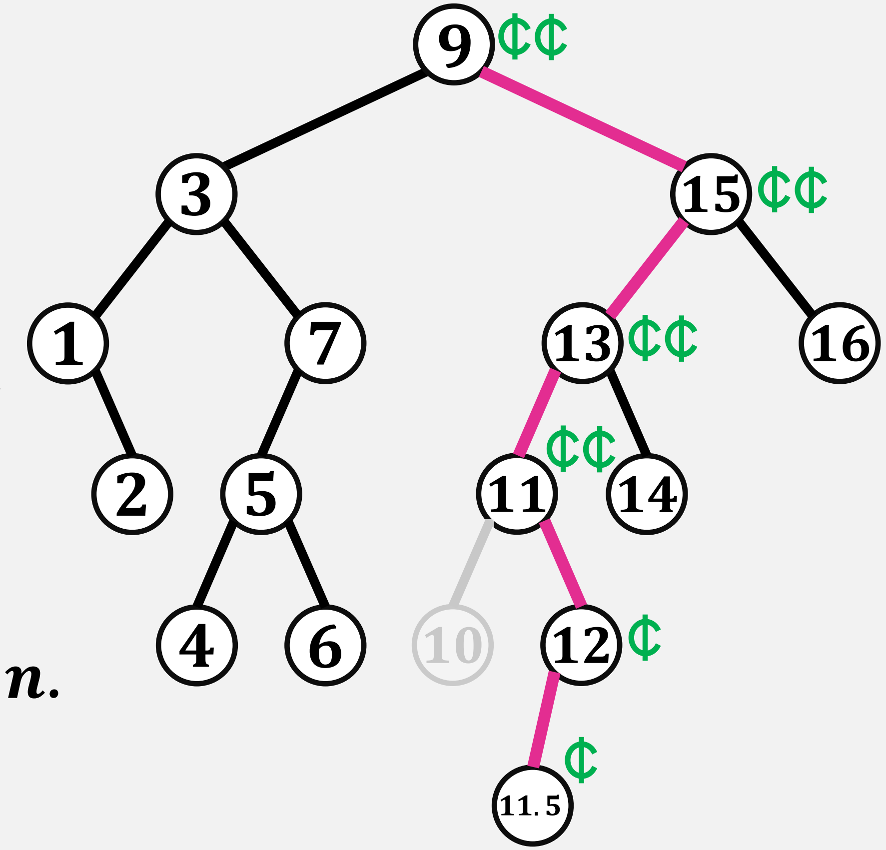
Credit scheme:

- each node stores a number of credits
- during an **insertion** or **deletion** of $u$, we give one credit to each node on the path to $u$.
- during a deletion we also store an additional credit

Invariant:

1. The number of $q$-credits saved up is at least $q - n$.
2. The number of credits on any node $u$ is at least

$$|\text{size}(u.\text{left}) - \text{size}(u.\text{right})| - 1$$




¢¢

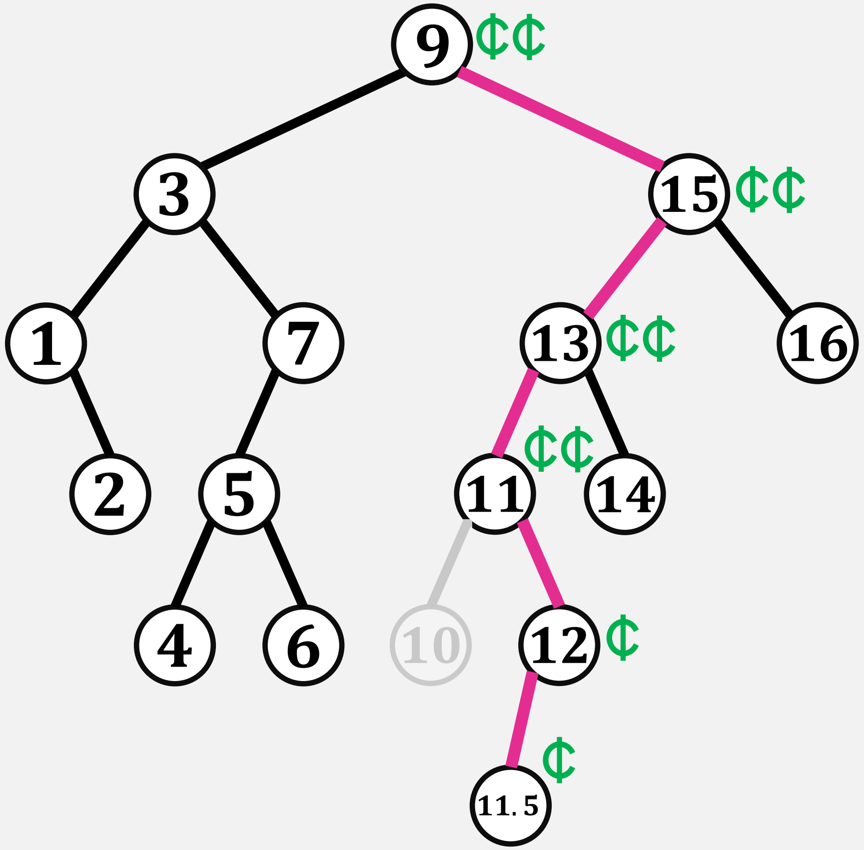
$n \leq q \leq 2n$

# Analysis

Invariant:

1. The number of $\boldsymbol{q}$-credits saved up is at least $\boldsymbol{q-n}$.
2. The number of credits on any node $\boldsymbol{u}$ is at least

$$|\text{size}(\boldsymbol{u}.\text{left}) - \text{size}(\boldsymbol{u}.\text{right})| - \mathbf{1}$$




If we call rebuild($\boldsymbol{u}$) during a **deletion**, it is because we decremented $\boldsymbol{n}$ and now $\boldsymbol{q > 2n}$.
But we have $\boldsymbol{q-n}$ $\boldsymbol{q}$-credits stored.

$$\boldsymbol{q - n > 2n - n = n}$$

We use these to pay for the $\boldsymbol{O(n)}$ time it takes to rebuild the root.

¢¢

$$\boldsymbol{n \le q \le 2n}$$

# Analysis

Invariant:

1. The number of $\boldsymbol{q}$-credits saved up is at least $\boldsymbol{q} - \boldsymbol{n}$.
2. The number of credits on any node $\boldsymbol{u}$ is at least
$$|\text{size}(\boldsymbol{u}.\text{left}) - \text{size}(\boldsymbol{u}.\text{right})| - \mathbf{1}$$

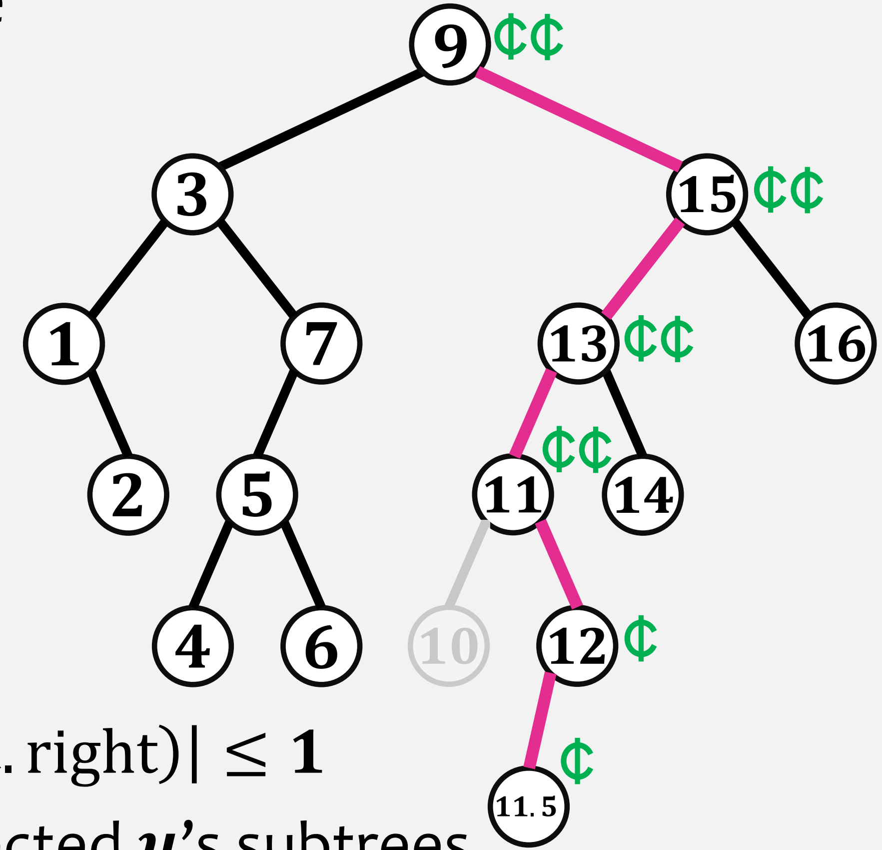
$\boldsymbol{O}(\text{size}(\boldsymbol{u}))$

If we call rebuild($\boldsymbol{u}$) during an **insertion**, it is because $\boldsymbol{u}$ is a scapegoat. Assume that path goes to the left:

$$\frac{\text{size}(\boldsymbol{u}.\text{left})}{\text{size}(\boldsymbol{u})} > \frac{\mathbf{2}}{\mathbf{3}} \qquad \text{size}(\boldsymbol{u}.\text{left}) > \mathbf{2}\text{size}(\boldsymbol{u}.\text{right})$$

$$\text{size}(\boldsymbol{u}) = \mathbf{1} + \text{size}(\boldsymbol{u}.\text{left}) + \text{size}(\boldsymbol{u}.\text{right})$$

$$\text{size}(\boldsymbol{u}.\text{left}) - \text{size}(\boldsymbol{u}.\text{right}) > \frac{1}{3}\text{size}(\boldsymbol{u})$$




The last time a subtree containing $\boldsymbol{u}$ was rebuilt (or when $\boldsymbol{u}$ was inserted), we had $|\text{size}(\boldsymbol{u}.\text{left}) - \text{size}(\boldsymbol{u}.\text{right})| \leq \mathbf{1}$

The number of add/remove operations that have affected $\boldsymbol{u}$'s subtrees since then is at least

$$\frac{\mathbf{1}}{\mathbf{3}}\text{size}(\boldsymbol{u}) - \mathbf{1} \leq \#\text{ credits stored at } \boldsymbol{u}$$

$$\boldsymbol{n} \leq \overset{¢¢}{\boldsymbol{q}} \leq \mathbf{2}\boldsymbol{n}$$

# Analysis

- During an insertion or deletion, we give one credit to each node on the path to the inserted (or deleted) node $\boldsymbol{u}$.

  So we hand out at most $\log_{3/2} \boldsymbol{q} \leq \log_{3/2} \boldsymbol{m}$ credits per operation.

- During a deletion we also store an additional credit with $\boldsymbol{q}$ .

Thus, in total we give out at most $\boldsymbol{O}(\boldsymbol{m} \log \boldsymbol{m})$ credits.

# Theorem 8.1

A **ScapegoatTree** implements the **SSet** interface. Ignoring the cost of rebuild($\boldsymbol{u}$) operations, a **ScapegoatTree** supports the operations add($\boldsymbol{x}$), remove($\boldsymbol{x}$), and find($\boldsymbol{x}$) in $\boldsymbol{O}(\log \boldsymbol{n})$ time per operation.
Furthermore, beginning with an empty **ScapegoatTree**, any sequence of $\boldsymbol{m}$ add($\boldsymbol{x}$) and remove($\boldsymbol{x}$) operations results in a total of $\boldsymbol{O}(\boldsymbol{m}\log \boldsymbol{m})$ time spent during all calls to rebuild($\boldsymbol{u}$).