Advanced Algorithmics and Artificial Intelligence

Olivier Baudon

Université de Bordeaux

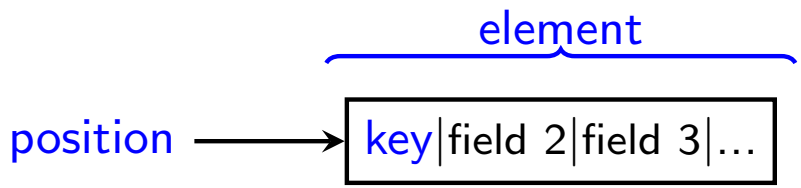
March 2021

- Advanced Data Structures

- Advanced Data Structures
  - Dynamic Sets
  - List Structures
  - Tree structures

Dynamic Sets: the abstract data type

To manage variable sets of elements of a same type, identified by a key:



we need to implement the following primitive functions:

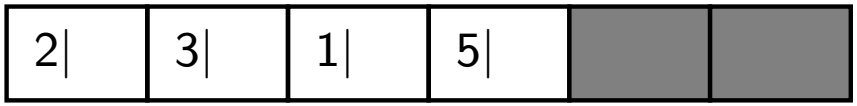
- `INSERT(element)`
- `DELETE(position)`
- `SEARCH(key)`

We wish to minimize

- the running time of the primitive functions
- the additional memory used to store informations.

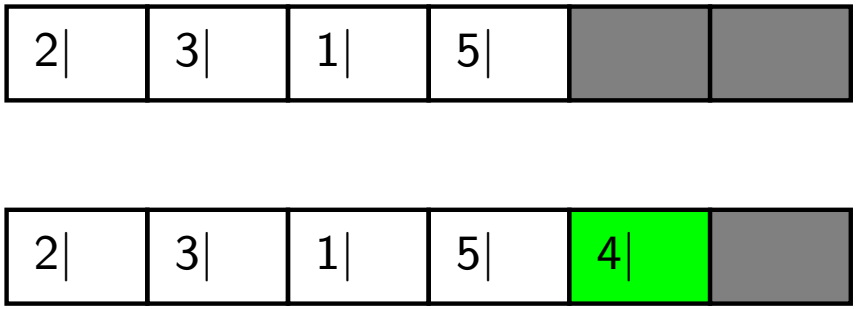
- Advanced Data Structures
  - Dynamic Sets
  - List Structures
  - Tree structures

Simple Arrays



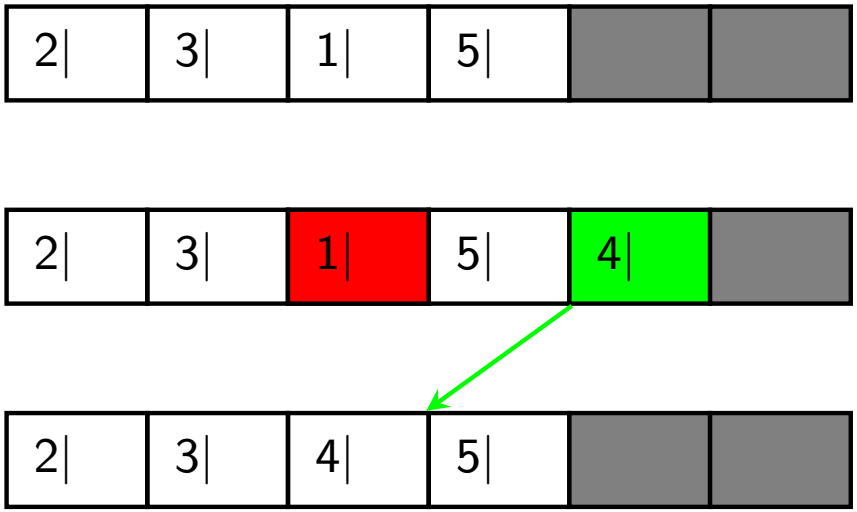
- Memory use : the expected maximal size of the set

Simple Arrays



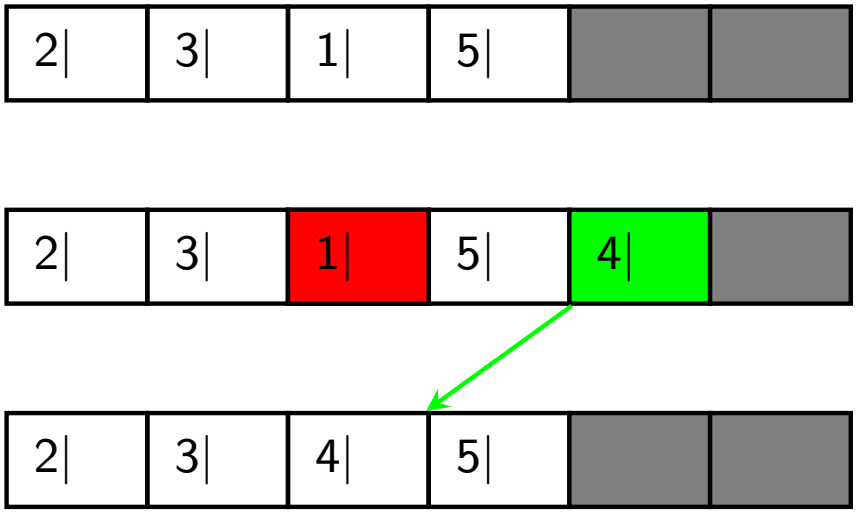
- Memory use : the expected maximal size of the set
- `INSERT(element)`: $\Theta(1)$

Simple Arrays



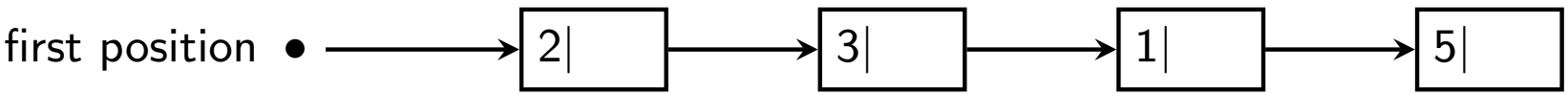
- Memory use : the expected maximal size of the set
- `INSERT(element)`: $\Theta(1)$
- `DELETE(position=index)`: $\Theta(1)$

Simple Arrays



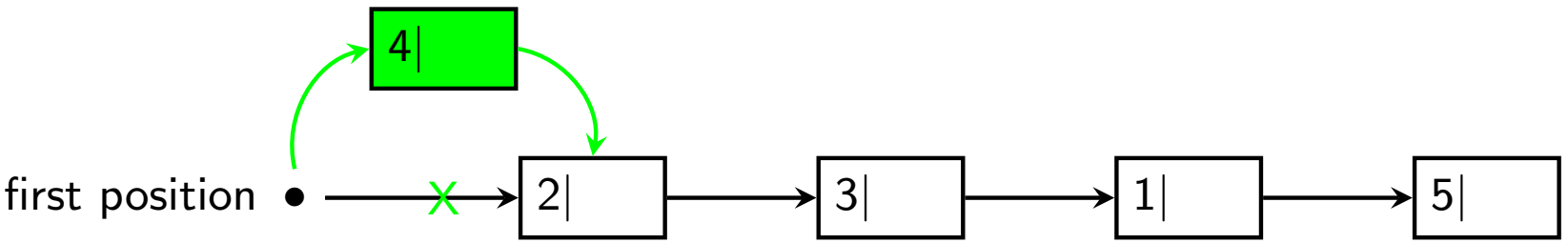
- Memory use : the expected maximal size of the set
- `INSERT(element)`: $\Theta(1)$
- `DELETE(position=index)`: $\Theta(1)$
- `SEARCH(key)`: $O(n)$

Linked Lists



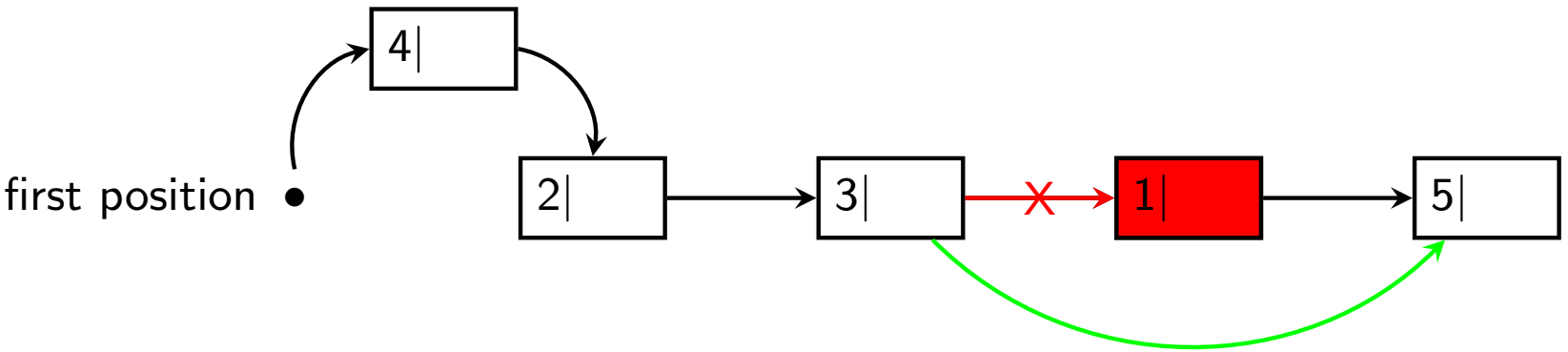
- Extra memory use : proportional to the size of the set

Linked Lists



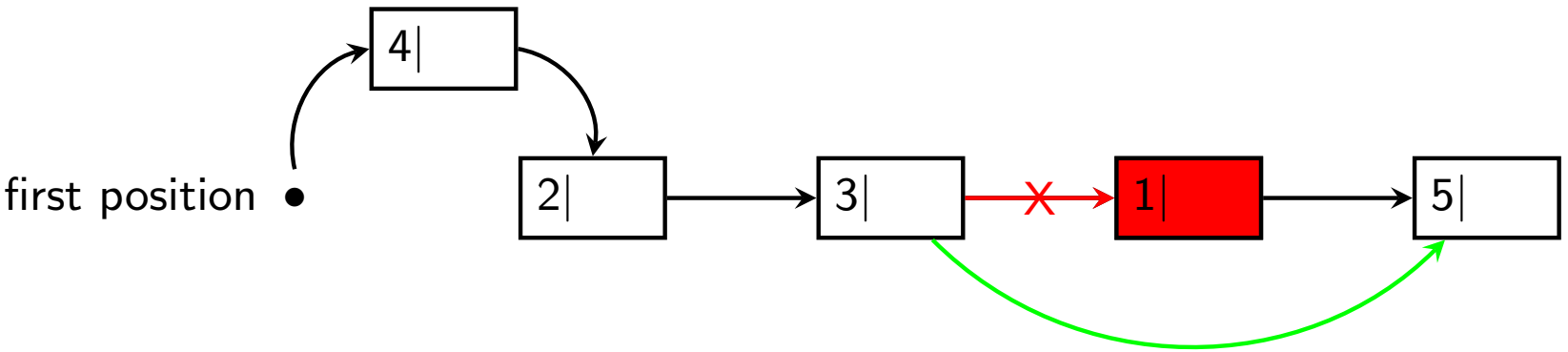
- Extra memory use : proportional to the size of the set
- `INSERT(element)`: $\Theta(1)$

Linked Lists



- Extra memory use : proportional to the size of the set
- `INSERT(element)`: $\Theta(1)$
- `DELETE(position=location of previous item)`: $\Theta(1)$

Linked Lists

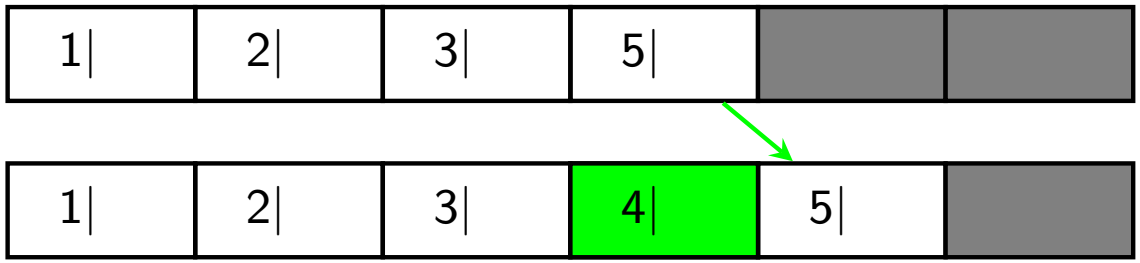


- Extra memory use : proportional to the size of the set
- `INSERT(element)`: $\Theta(1)$
- `DELETE(position=location of previous item)`: $\Theta(1)$
- `SEARCH(key)`: $O(n)$

Key-Sorted Arrays

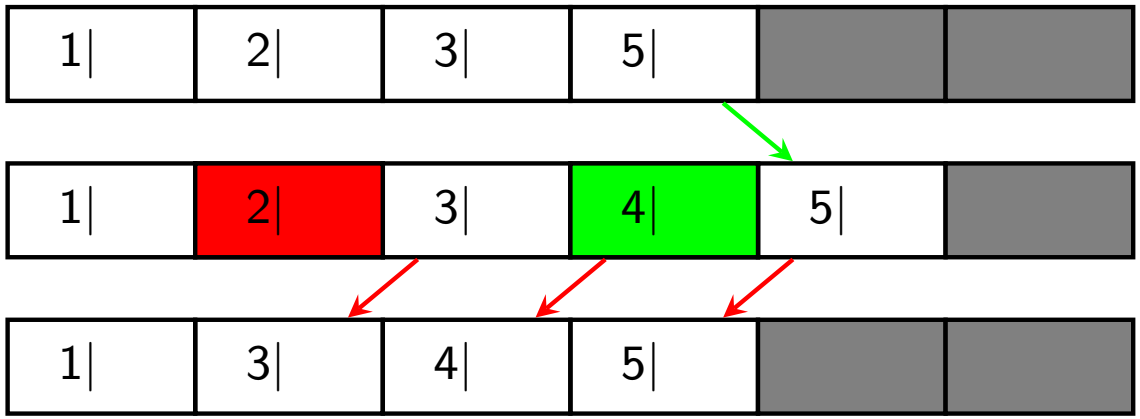
- Memory use : the expected maximal size of the set
- `SEARCH(key)`: $O(\log_2 n)$

Key-Sorted Arrays



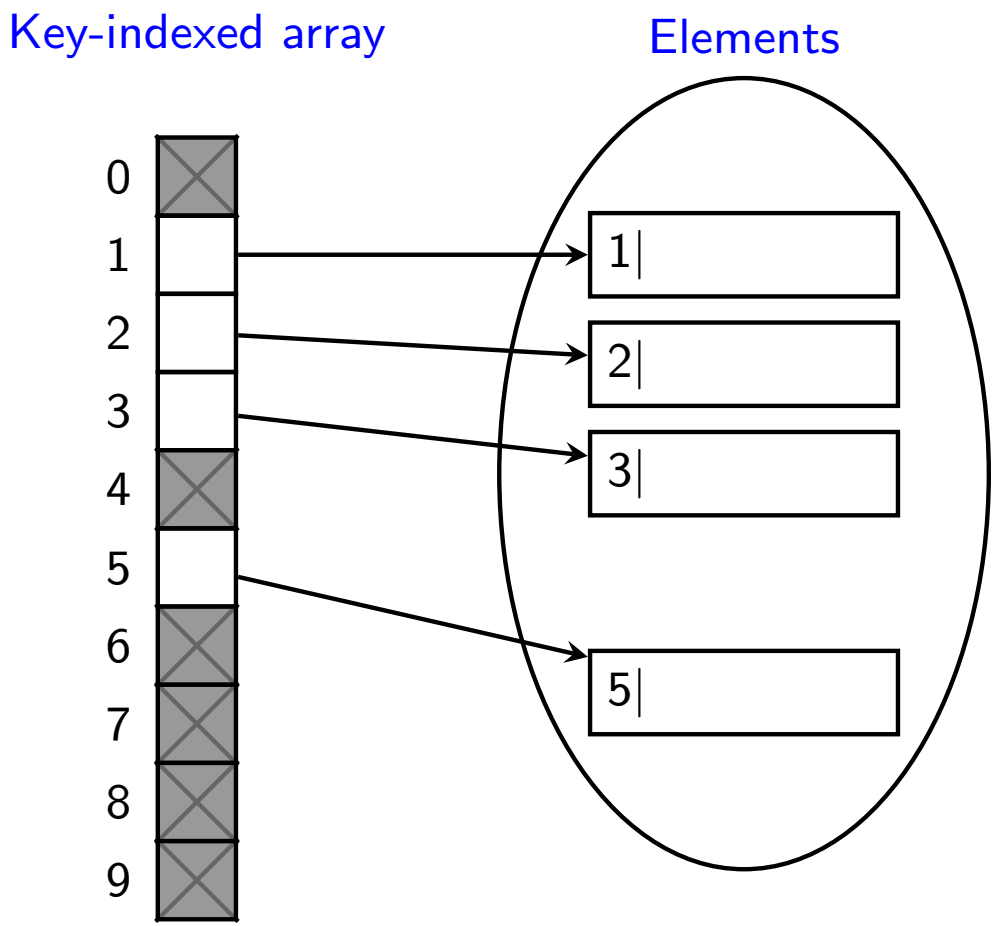
- Memory use : the expected maximal size of the set
- `SEARCH(key)`: $O(\log_2 n)$
- `INSERT(element)`: $O(n)$

Key-Sorted Arrays



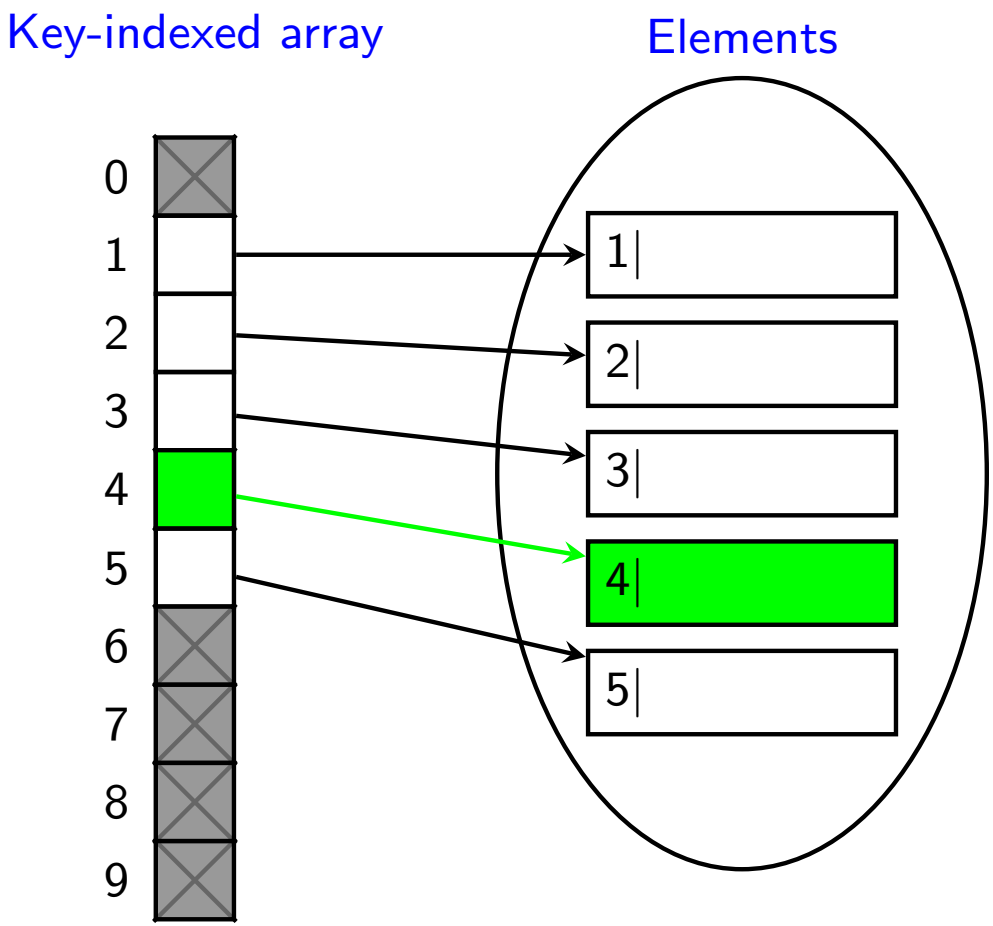
- Memory use : the expected maximal size of the set
- `SEARCH(key)`: $O(\log_2 n)$
- `INSERT(element)`: $O(n)$
- `DELETE(position=index)`: $O(n)$

Direct-Access Tables



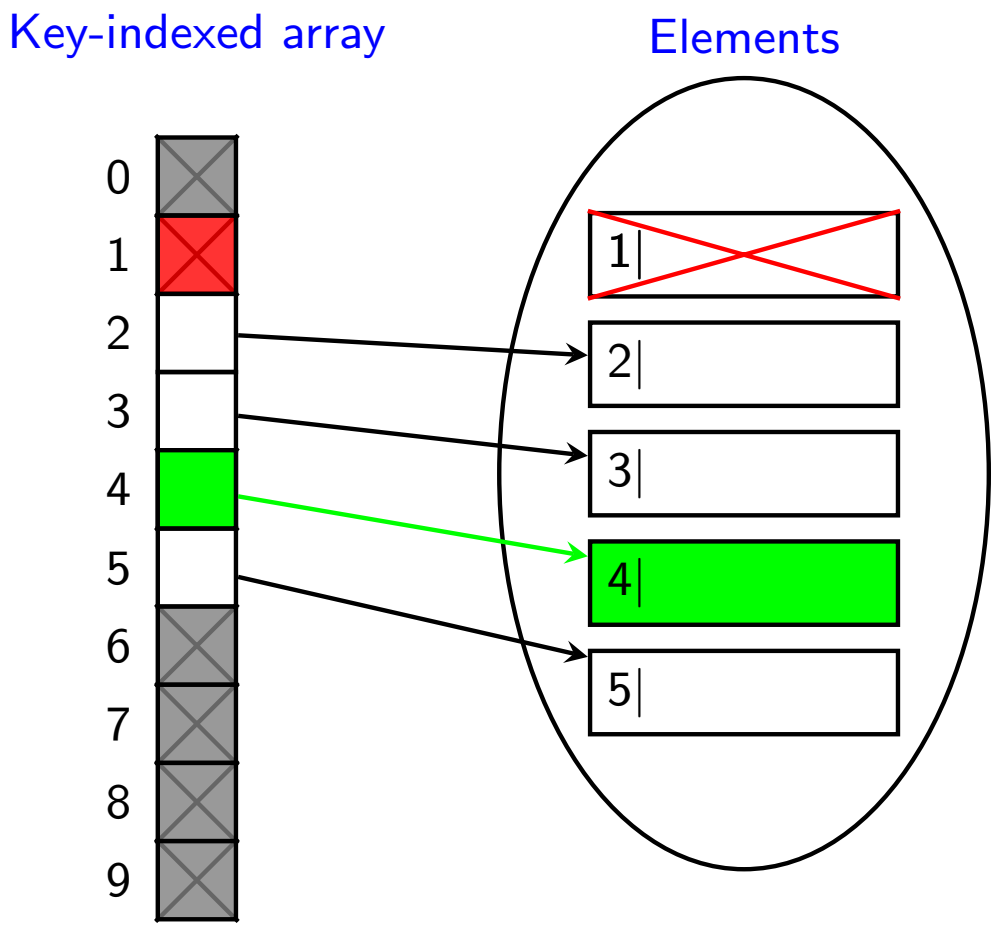
- `SEARCH(key)`: $\Theta(1)$

Direct-Access Tables



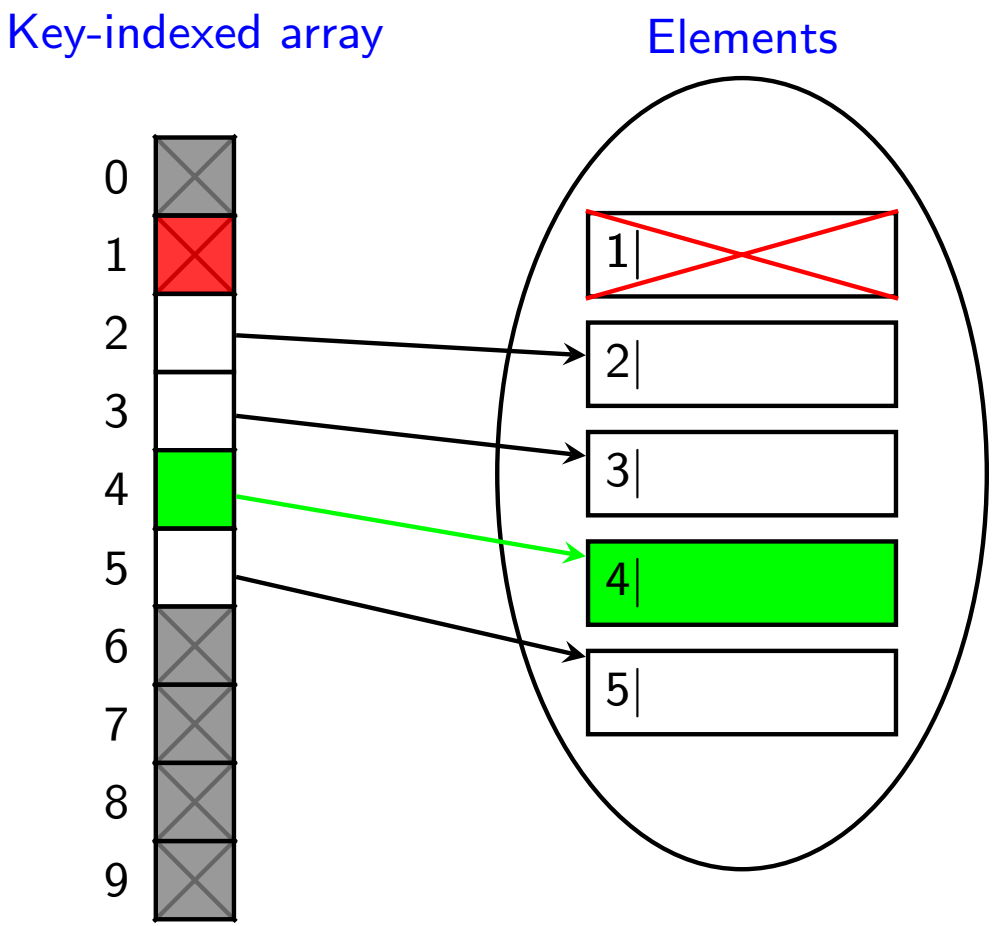
- `SEARCH(key)`: $\Theta(1)$, `INSERT(element)`: $\Theta(1)$

Direct-Access Tables



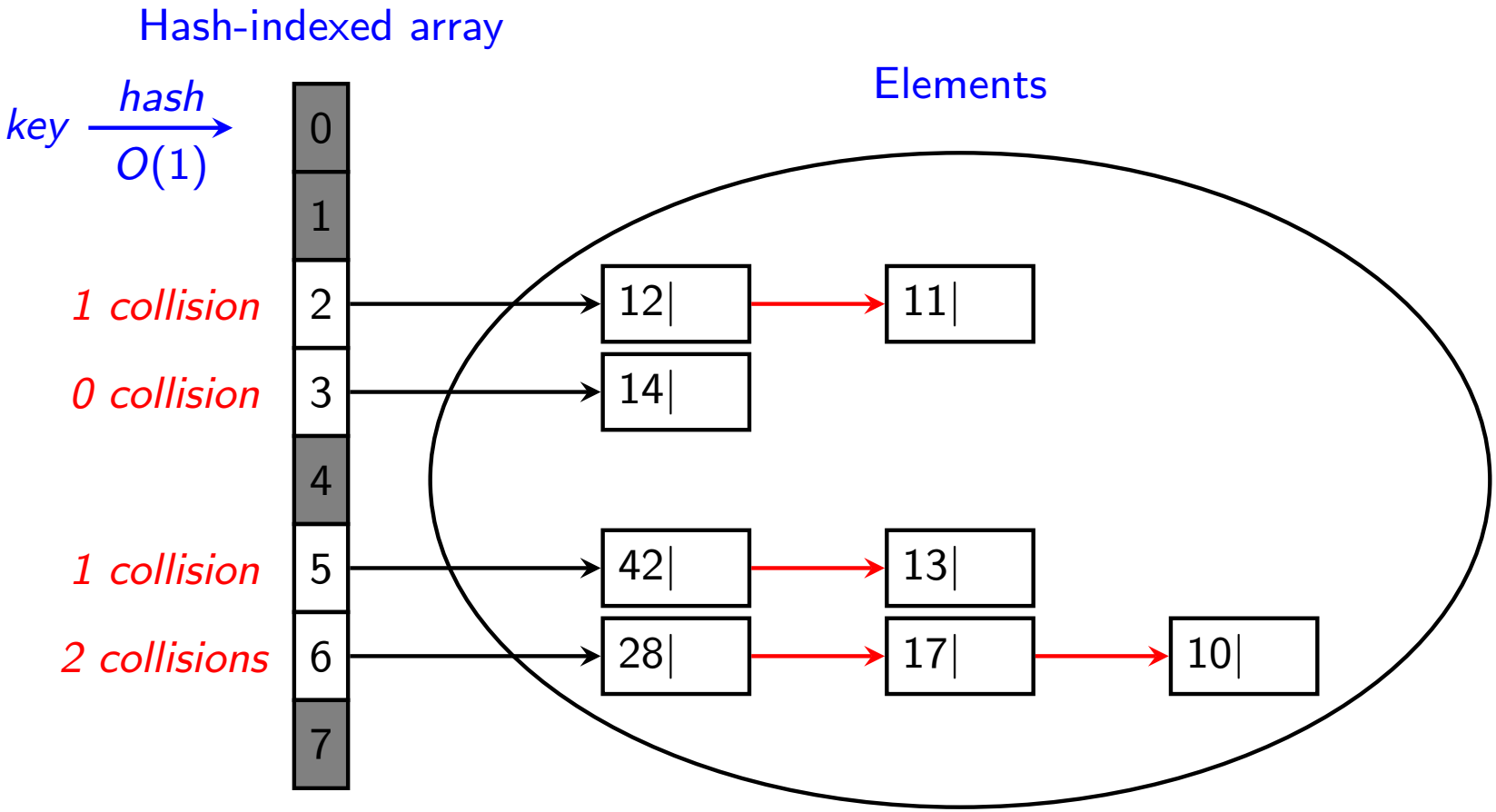
- `SEARCH(key)`: $\Theta(1)$, `INSERT(element)`: $\Theta(1)$, `DELETE(key)`: $\Theta(1)$

Direct-Access Tables



- `SEARCH(key)`: $\Theta(1)$, `INSERT(element)`: $\Theta(1)$, `DELETE(key)`: $\Theta(1)$
- Extra memory use: an array indexed by all possible keys
  (may be huge, if not infinite!)

Hash Tables



- Extra memory use: the (fixed) size of the hash table
- `INSERT(element)`, `DELETE(position)`: $\Theta(1)$
- `SEARCH(key)`: $O(1 + \texttt{maximal number of collisions})$

Applications: address books, symbol tables, dictionaries...

Hashing Techniques

A hash function must associate with each key k a value $h(k)$, $0 \leq h(k) < m$ where m is the size of the hash table.

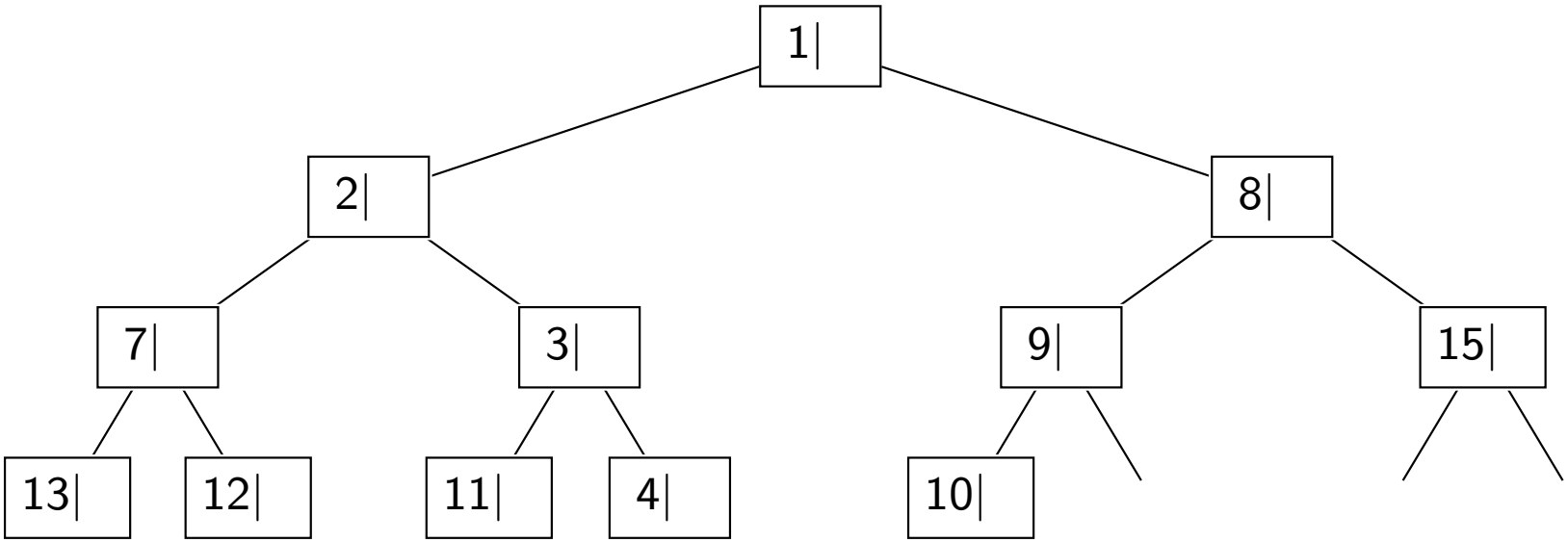
- **Division**: $h(k) = x \,\%\, m$
  where x is an interpretation of k as an integer:
  - typically $\sum 2^i b_i$ where b_i is the i-th bit of the key
  - more generally $\sum a_i x_i$ where the a_i are constant values, and the x_i represent fixed-size parts of the key.

  In practice, to minimize the maximal number of collisions, m should be a prime number "not too close to a power of 2".

- **Multiplication**: $h(k) = m(Ax - \lfloor Ax \rfloor)$ where A is a real constant, $0 < A < 1$.
  Advantage: in practice, no constraint on m.

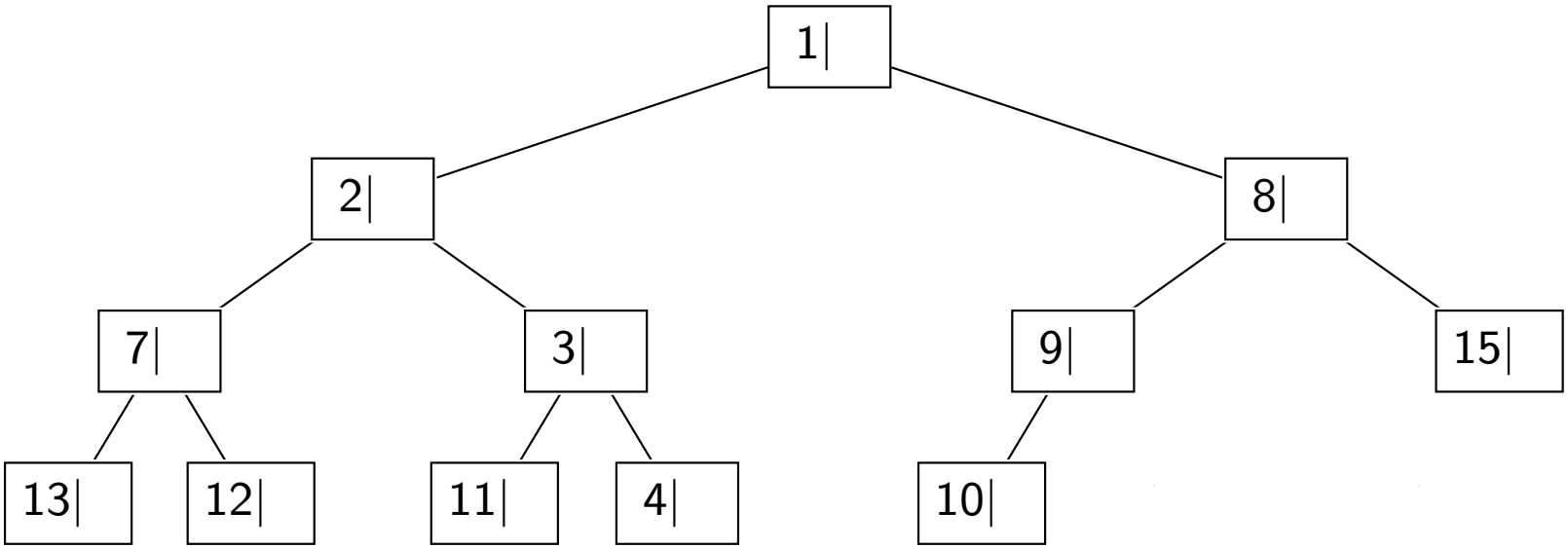
- Advanced Data Structures
  - Dynamic Sets
  - List Structures
  - Tree structures

Heaps: a partially ordered structure



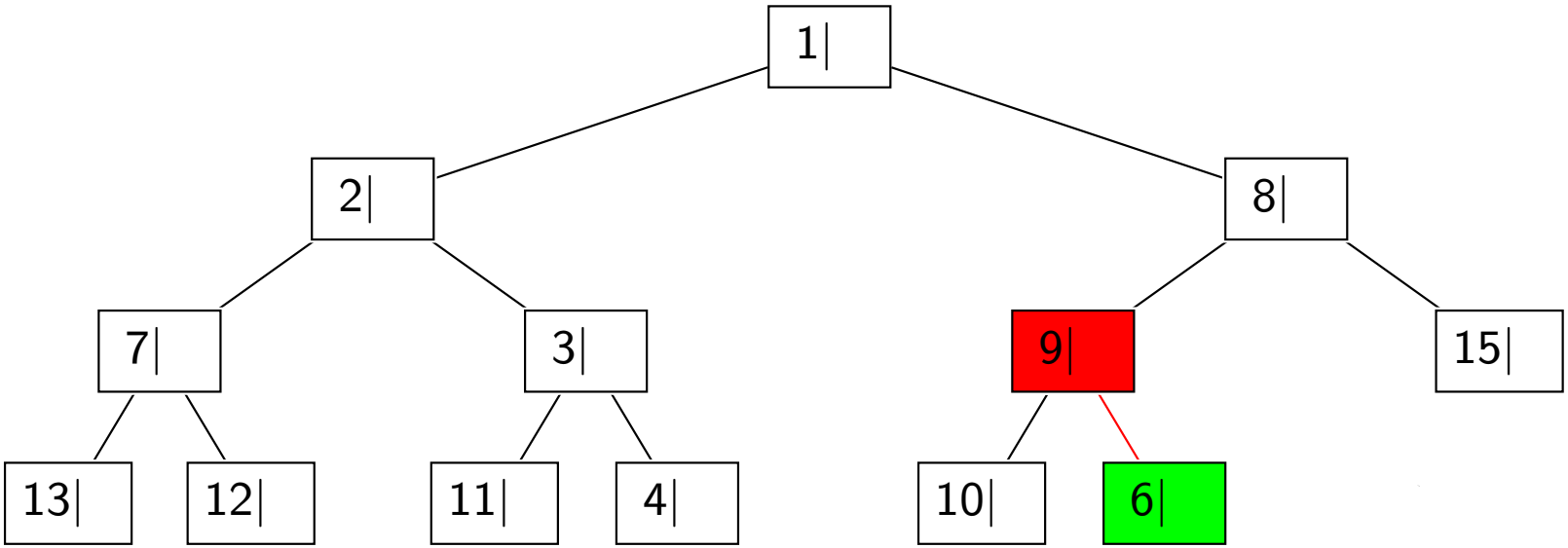
- Shape : n first nodes of the complete binary tree of height h ($2^h \leq n < 2^{h+1}$)
- Ordering : the key of a node $\leq$ the keys of its children

Heaps: Algorithms



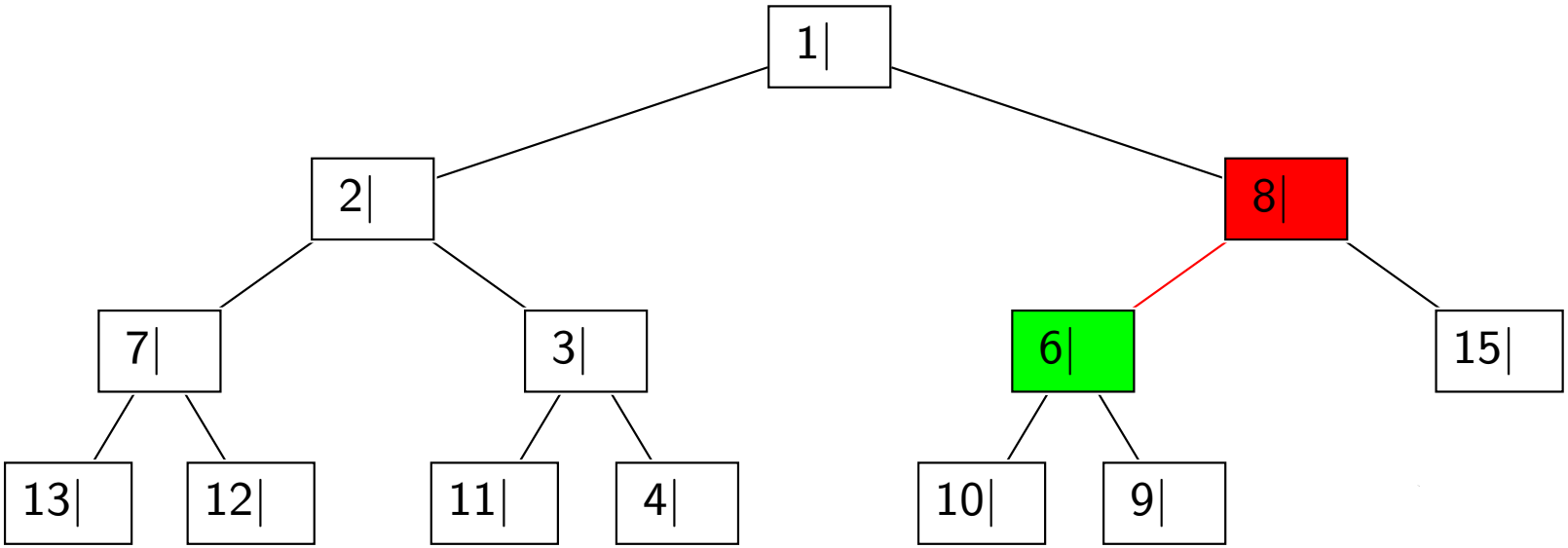
- `SEARCH(key)`: $O(n)$ – `MINIMUM()`: $\Theta(1)$

Heaps: Algorithms



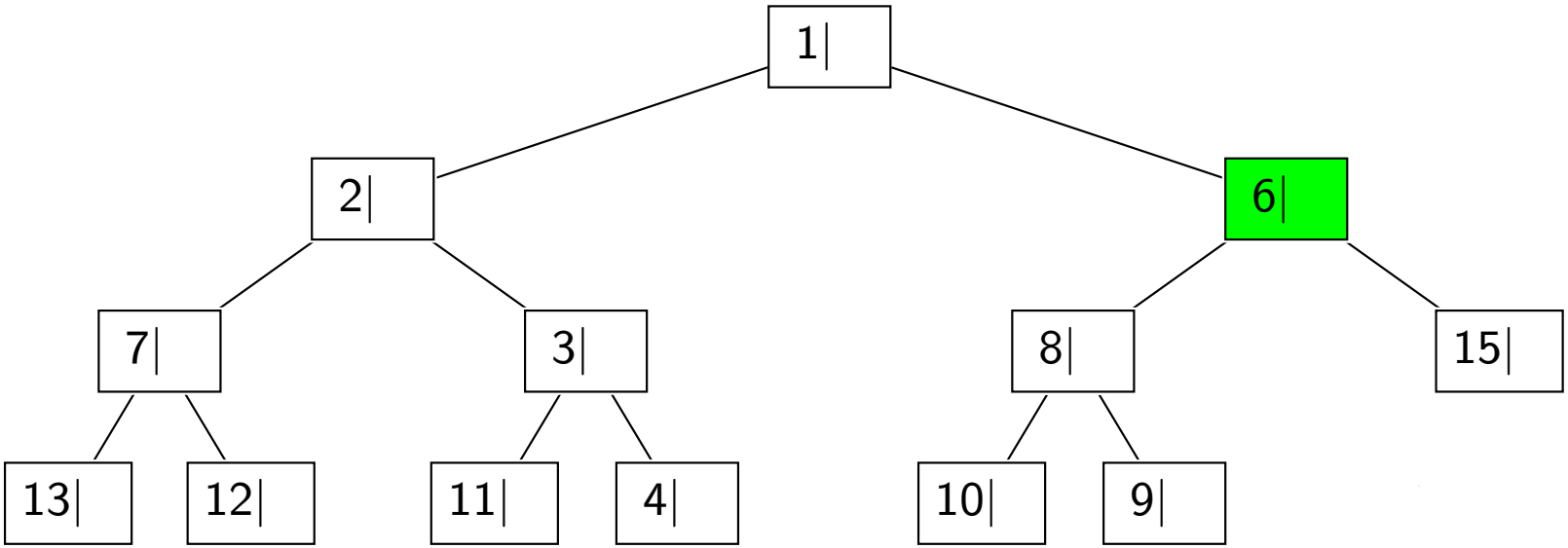
- `SEARCH(key)`: $O(n)$ – `MINIMUM()`: $\Theta(1)$
- `INSERT` : put it as the next free node and fix the heap recursively

Heaps: Algorithms



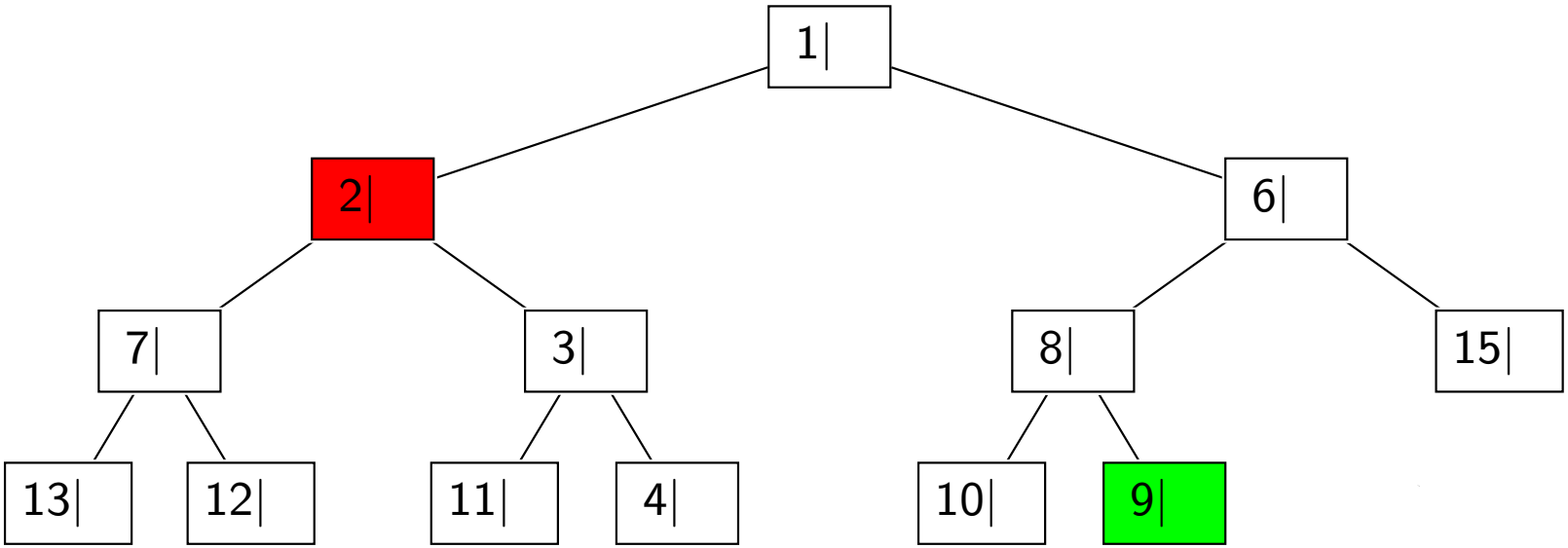
- `SEARCH(key)`: $O(n)$ – `MINIMUM()`: $\Theta(1)$
- `INSERT` : put it as the next free node and fix the heap recursively

Heaps: Algorithms



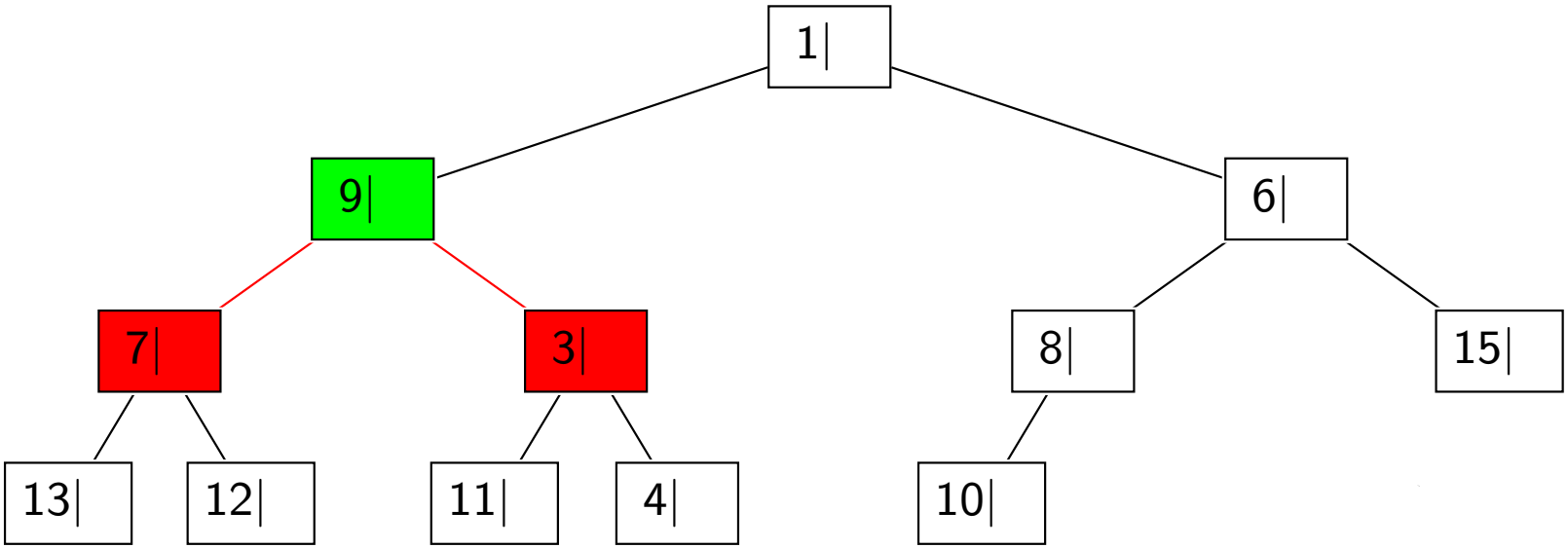
- `SEARCH(key)`: $O(n)$ – `MINIMUM()`: $\Theta(1)$
- `INSERT (element)`: $O(h) = O(\log_2(n))$

Heaps: Algorithms



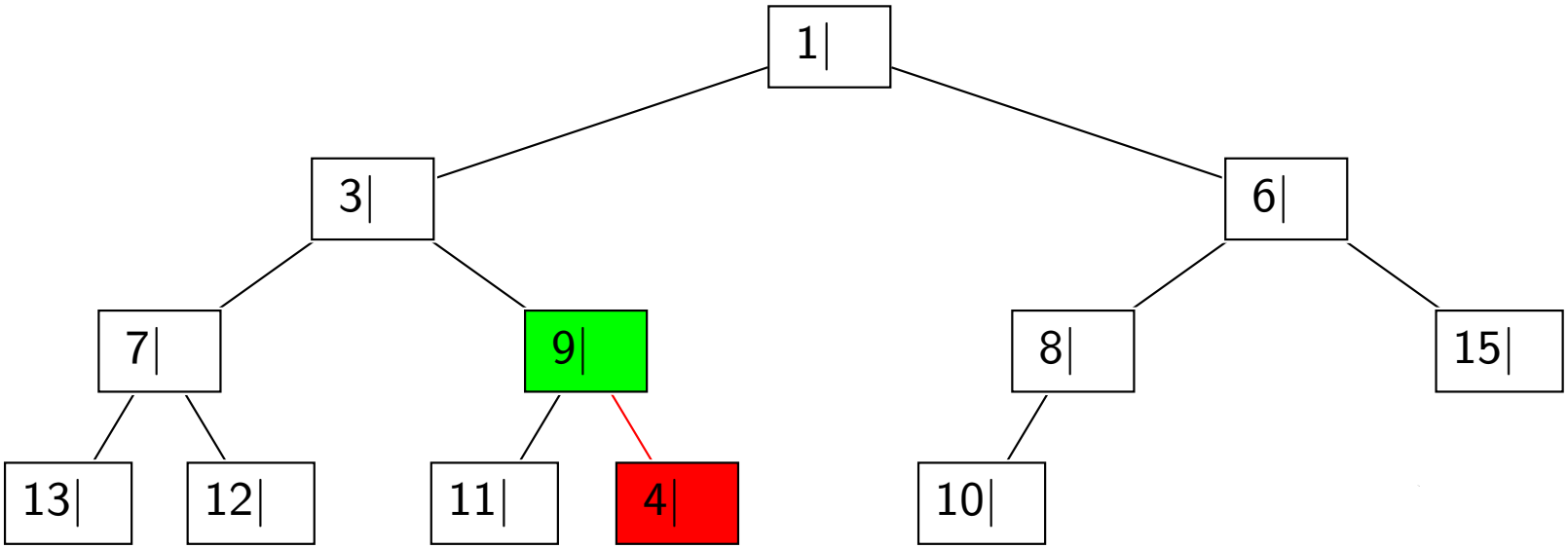
- `SEARCH(key)`: $O(n)$ – `MINIMUM()`: $\Theta(1)$
- `INSERT (element)`: $O(h) = O(\log_2(n))$
- `DELETE` : swap it with the last node (to be deleted)

Heaps: Algorithms



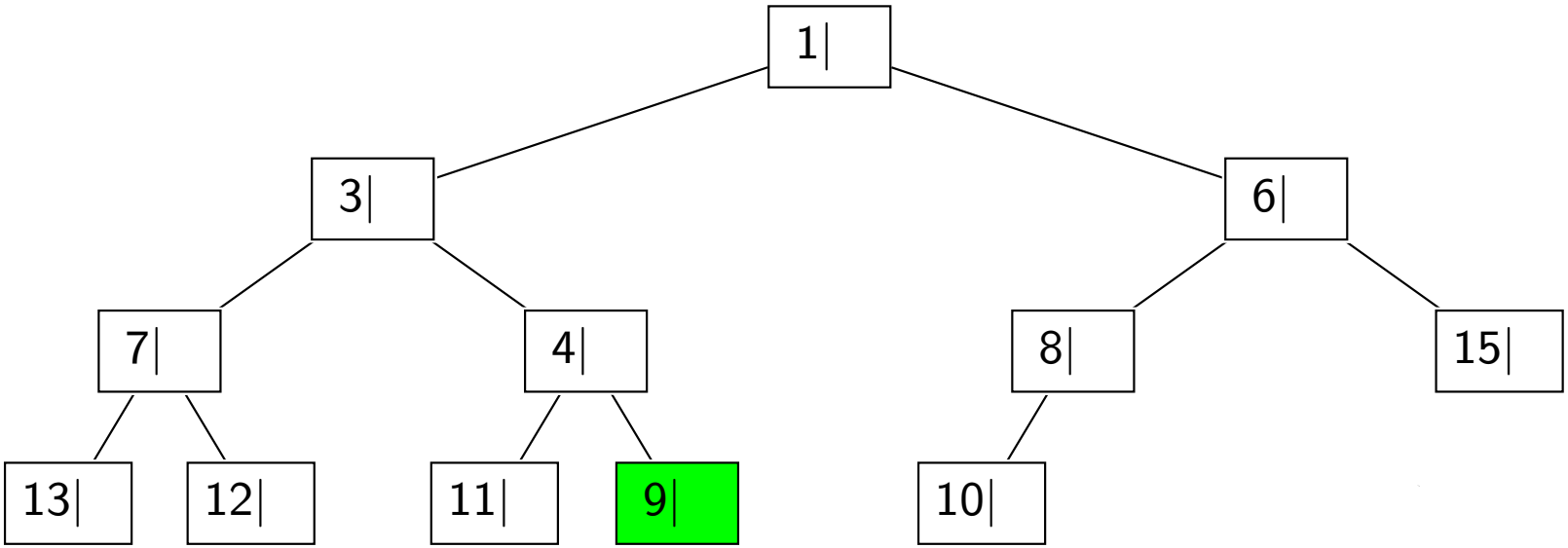
- `SEARCH(key)`: $O(n)$ – `MINIMUM()`: $\Theta(1)$
- `INSERT (element)`: $O(h) = O(\log_2(n))$
- `DELETE` : fix the heap recursively

Heaps: Algorithms



- `SEARCH(key)`: $O(n)$ – `MINIMUM()`: $\Theta(1)$
- `INSERT (element)`: $O(h) = O(\log_2(n))$
- `DELETE` : fix the heap recursively

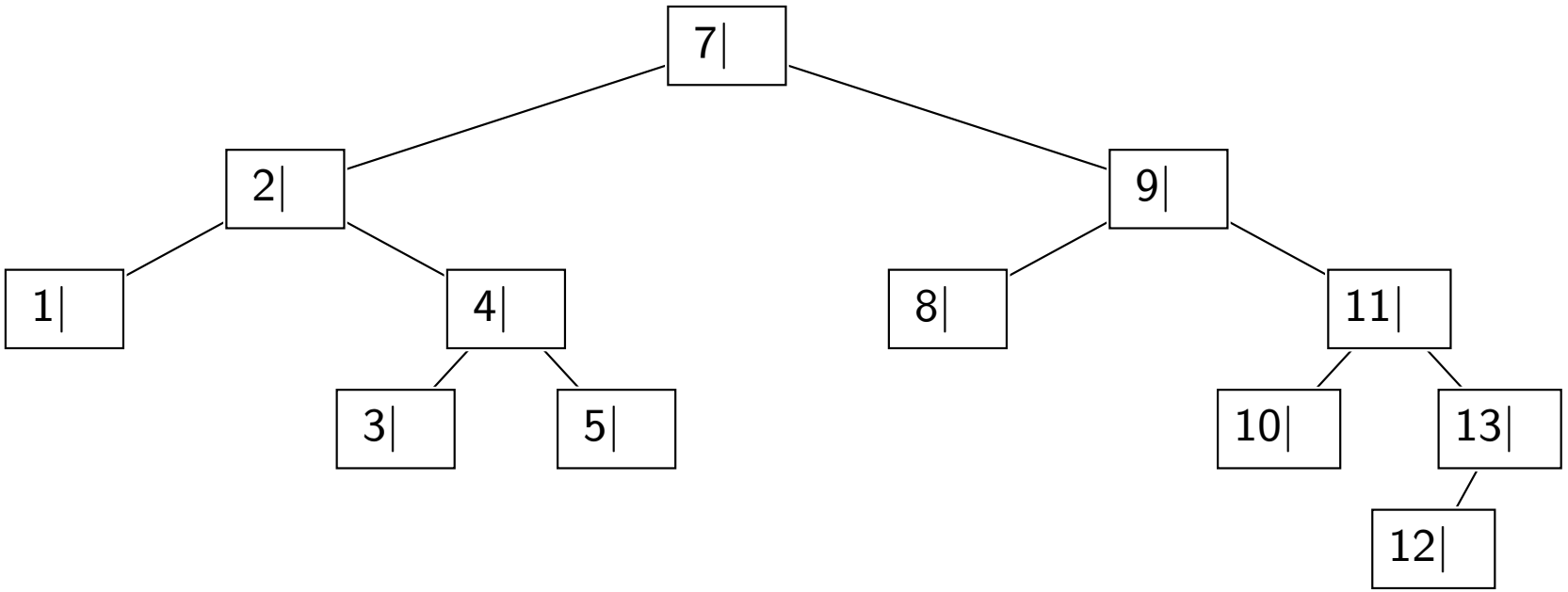
Heaps: Algorithms



- `SEARCH(key)`: $O(n)$ – `MINIMUM()`: $\Theta(1)$
- `INSERT (element)`: $O(h) = O(\log_2(n))$
- `DELETE (position)`: $O(h) = O(\log_2(n))$

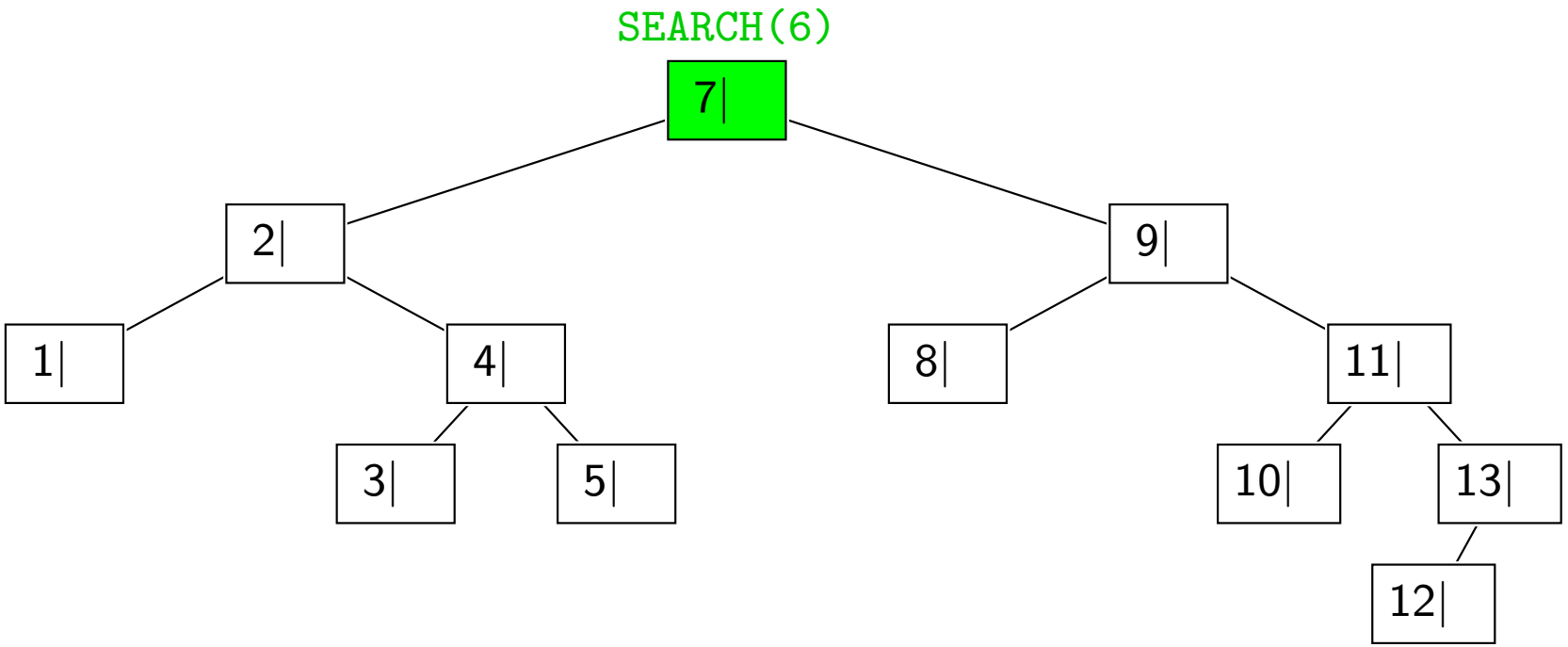
Application: priority queues.

Binary Search Trees: recursive partially ordered structures



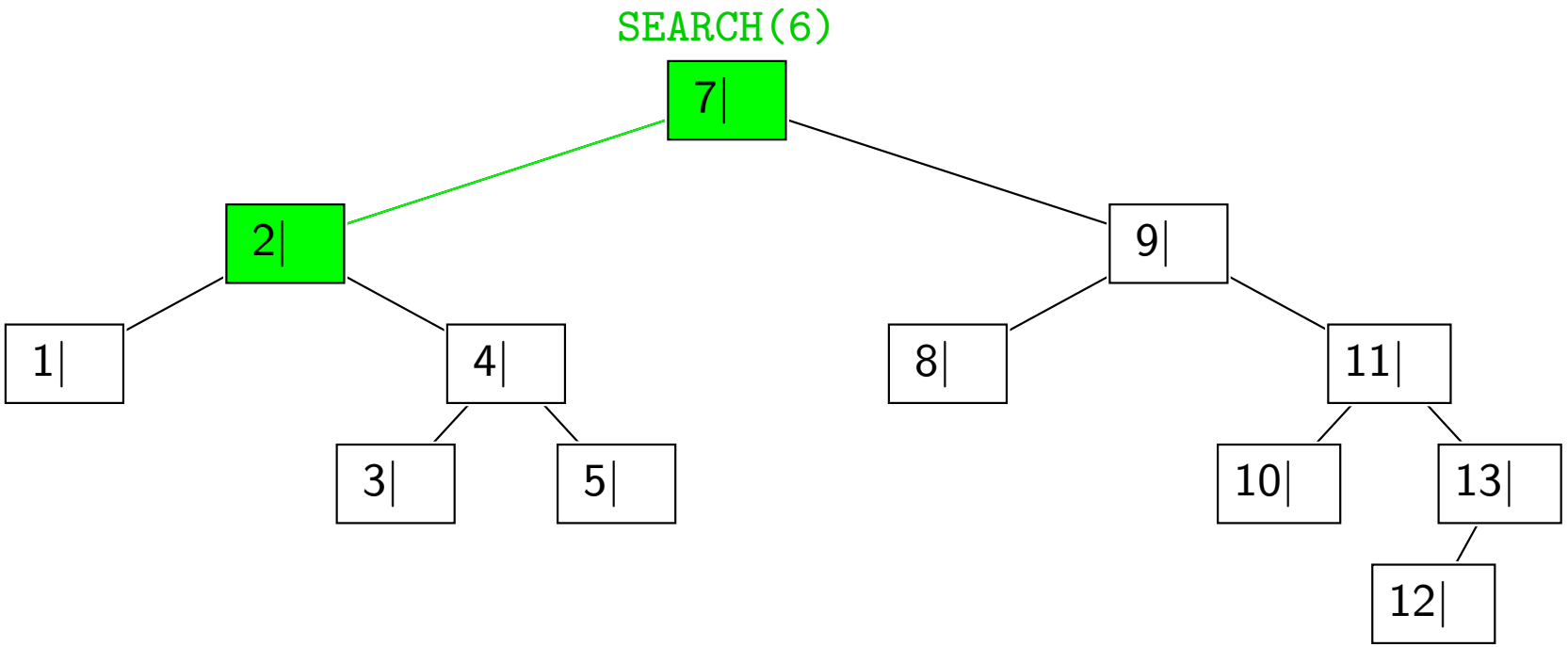
- Shape: either empty (not drawn), or a root node with two BST children
- Ordering: *key of the root* $\begin{cases} > \textit{ all keys stored in the left child} \\ < \textit{ all keys stored in the right child} \end{cases}$

Binary Search Trees: Algorithms



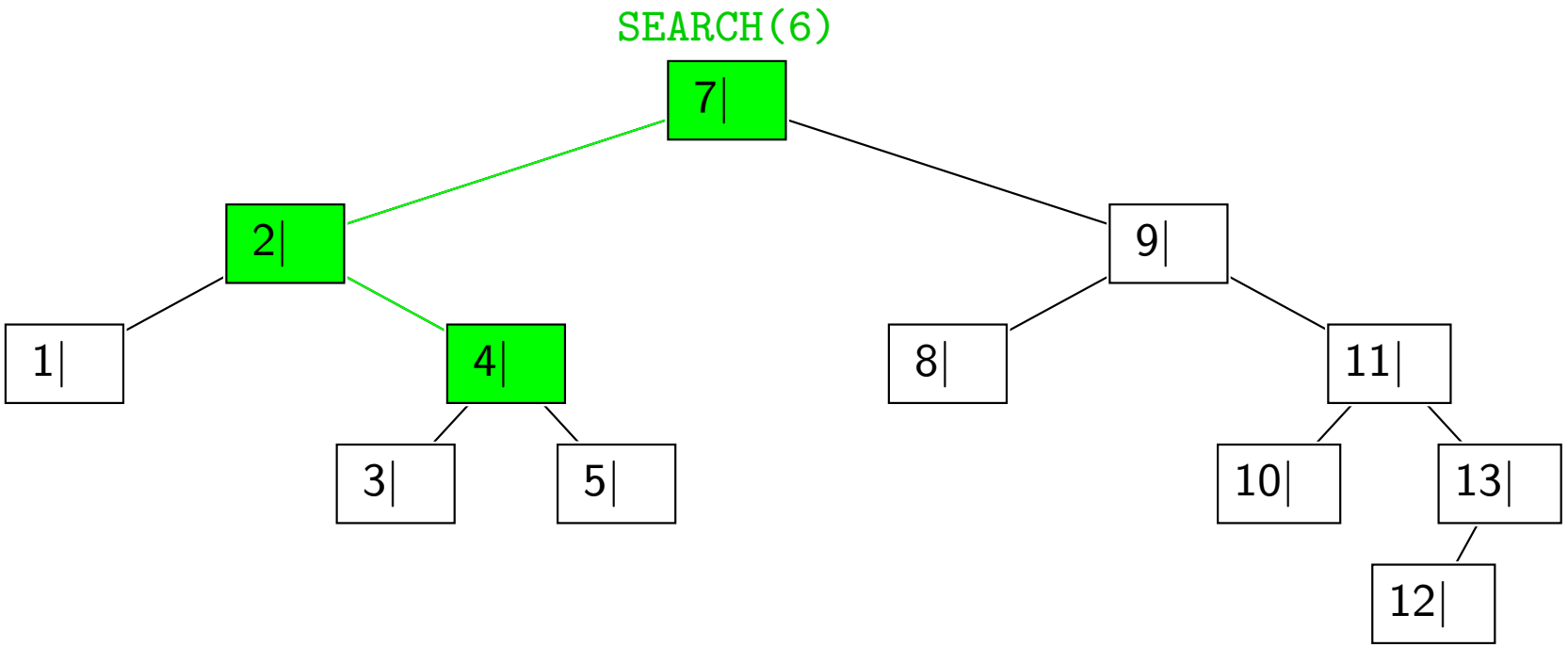
- SEARCH

Binary Search Trees: Algorithms



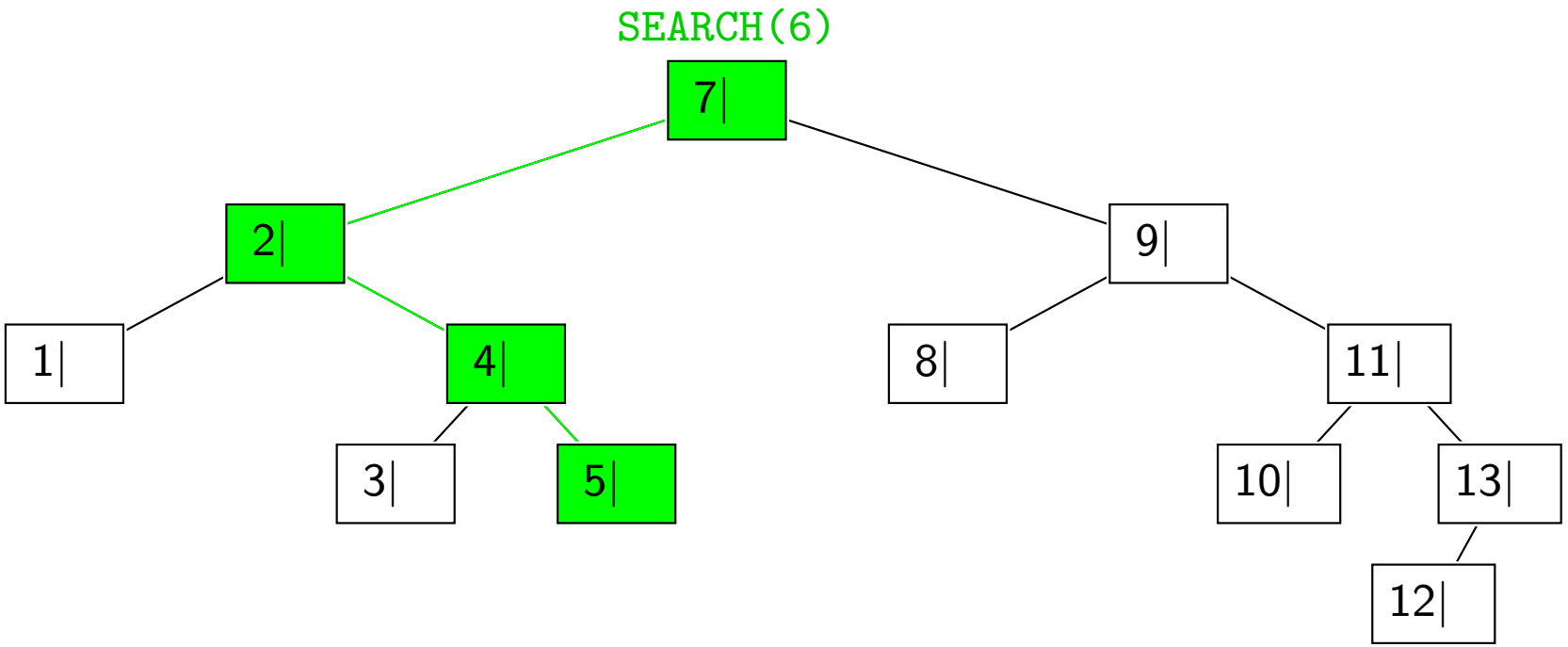
- `SEARCH`

Binary Search Trees: Algorithms



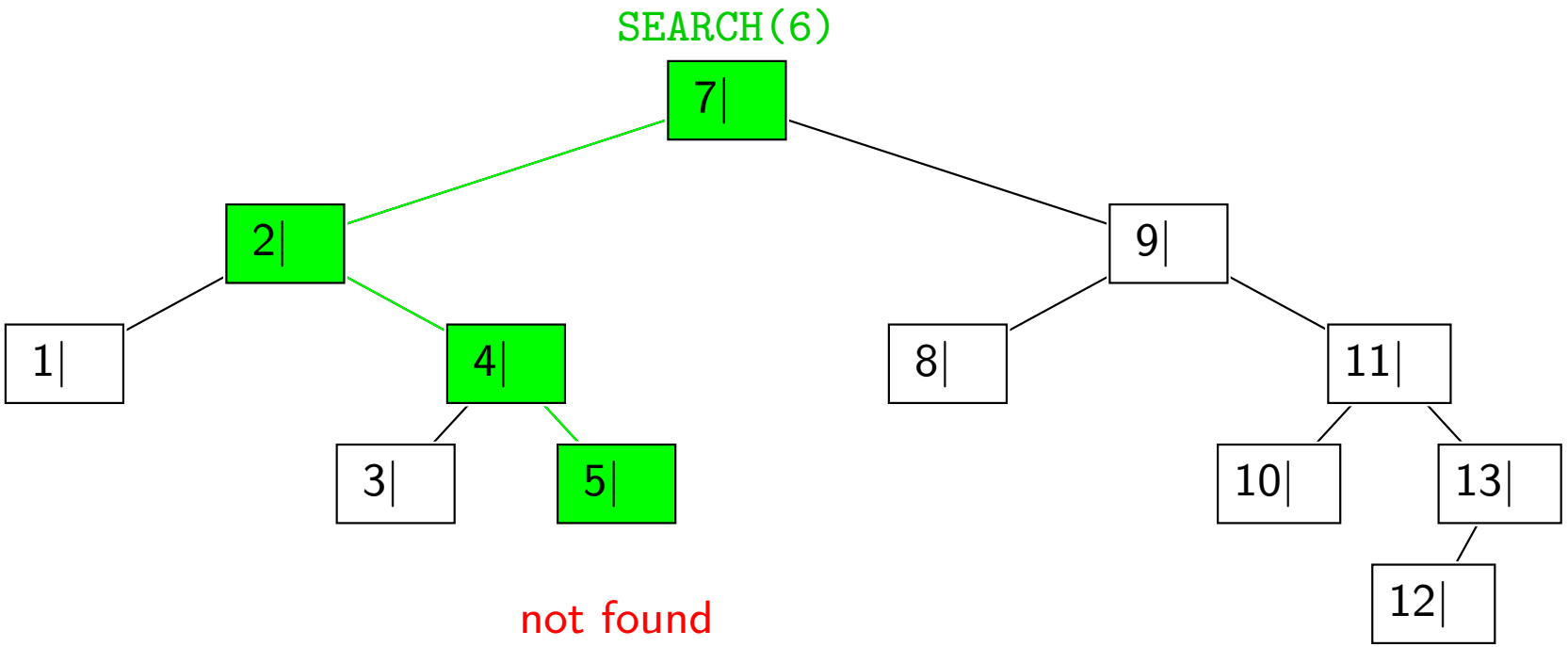
- SEARCH

Binary Search Trees: Algorithms



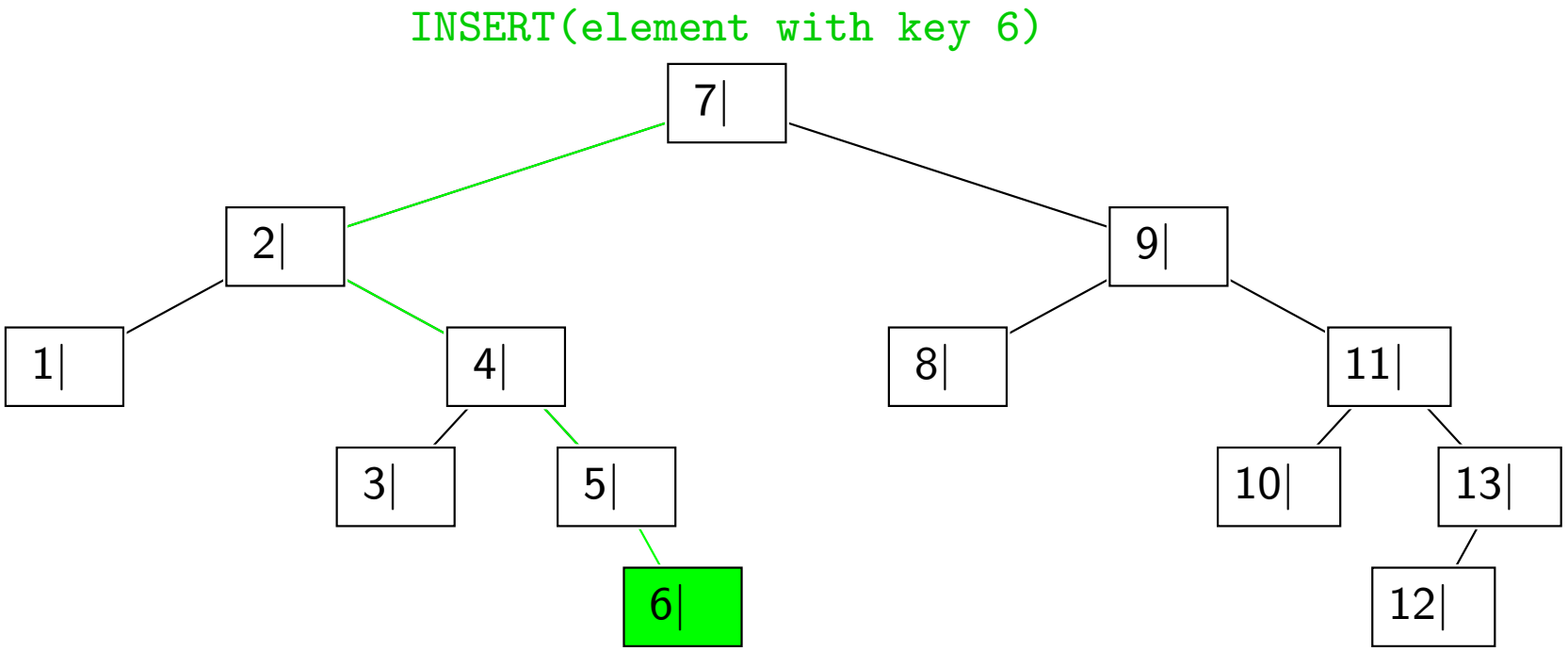
- SEARCH

Binary Search Trees: Algorithms



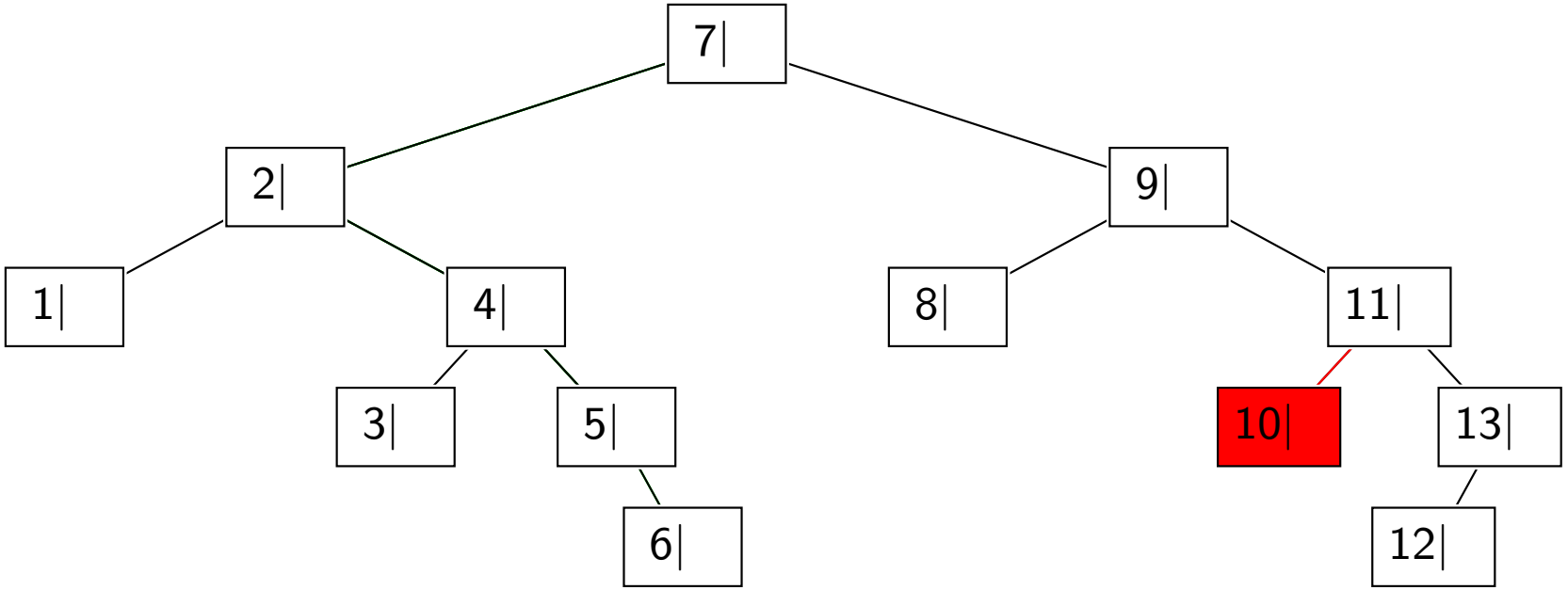
- `SEARCH(key)`: $O(h)$

Binary Search Trees: Algorithms



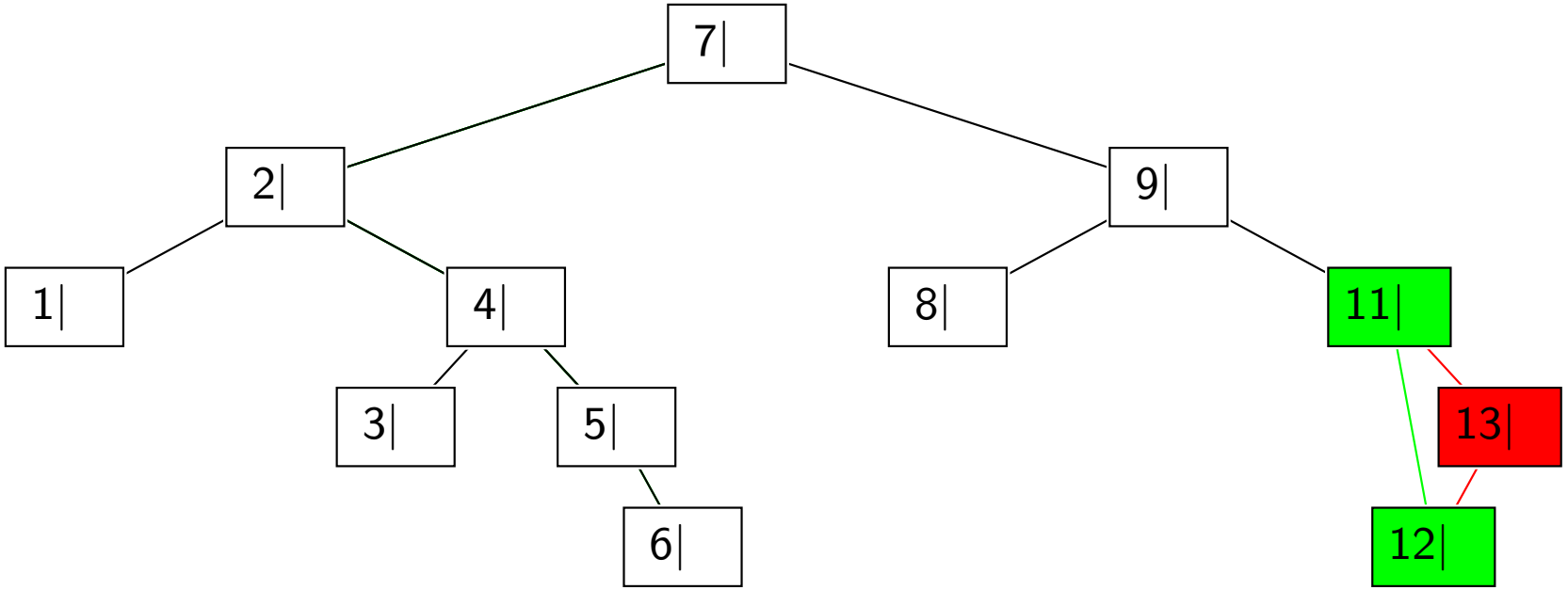
- `SEARCH(key)`: $O(h)$ – `INSERT(element)`: $O(h)$

Binary Search Trees: Algorithms



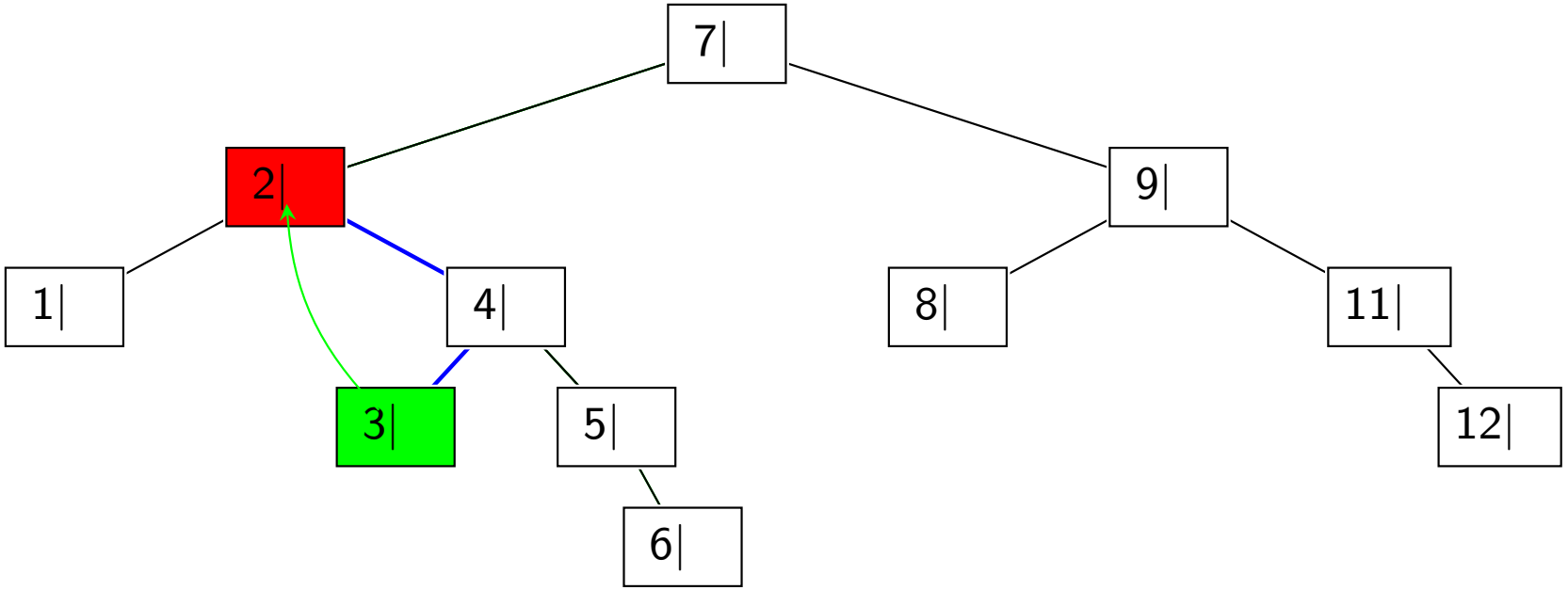
- `SEARCH(key)`: $O(h)$ – `INSERT(element)`: $O(h)$
- `DELETE`:
  - If the node has no child, remove it.

Binary Search Trees: Algorithms



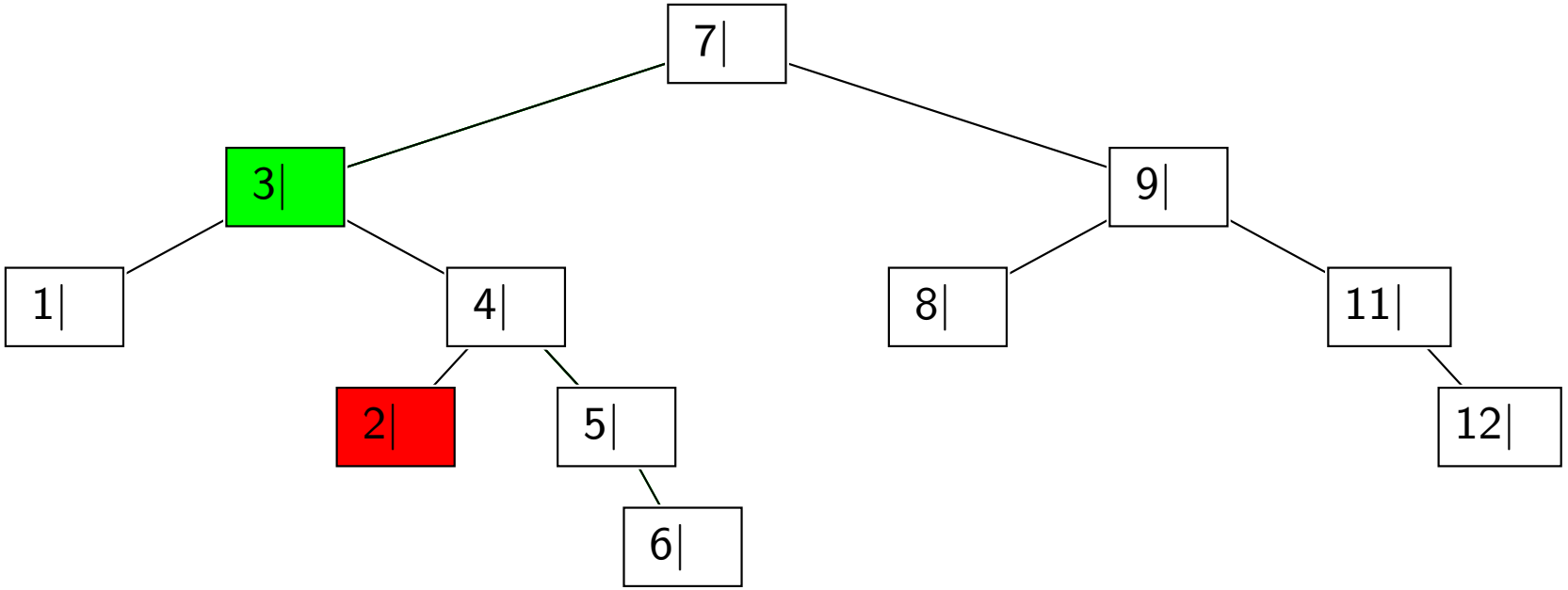
- `SEARCH(key)`: $O(h)$ – `INSERT(element)`: $O(h)$
- `DELETE`:
  - If the node has no child, remove it.
  - If the node has 1 child, link the child to the parent.

Binary Search Trees: Algorithms



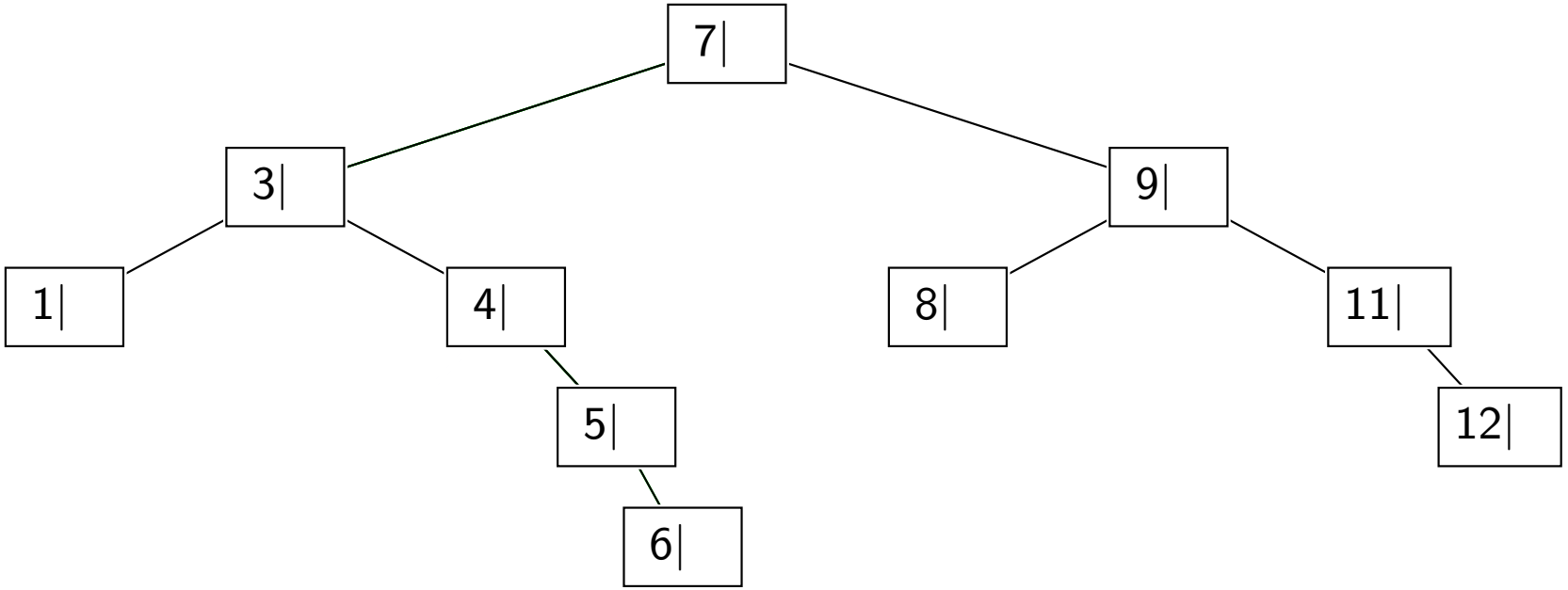
- `SEARCH(key)`: $O(h)$ – `INSERT(element)`: $O(h)$
- `DELETE`:
  - If the node has no child, remove it.
  - If the node has 1 child, link the child to the parent.
  - If the node has 2 children, swap it with its successor

Binary Search Trees: Algorithms



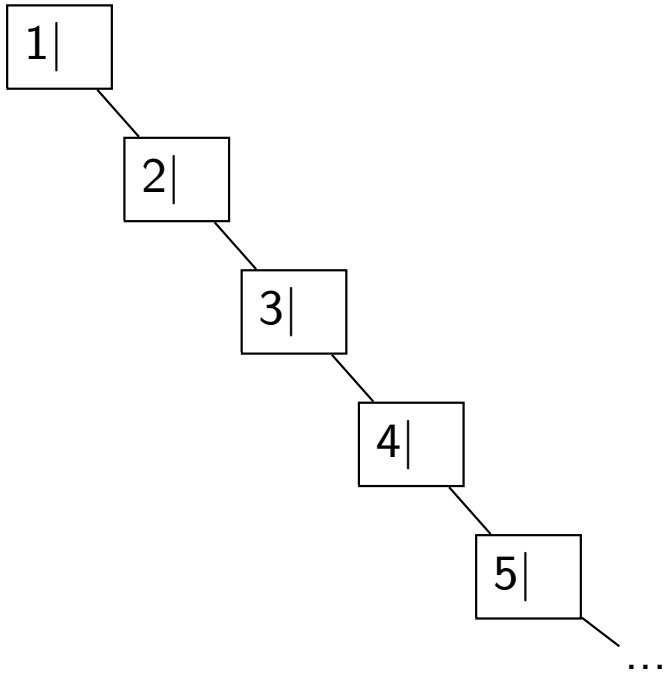
- `SEARCH(key)`: $O(h)$ – `INSERT(element)`: $O(h)$
- `DELETE`:
  - If the node has no child, remove it.
  - If the node has 1 child, link the child to the parent.
  - If the node has 2 children, swap it with its successor, and remove the successor.

Binary Search Trees: Algorithms



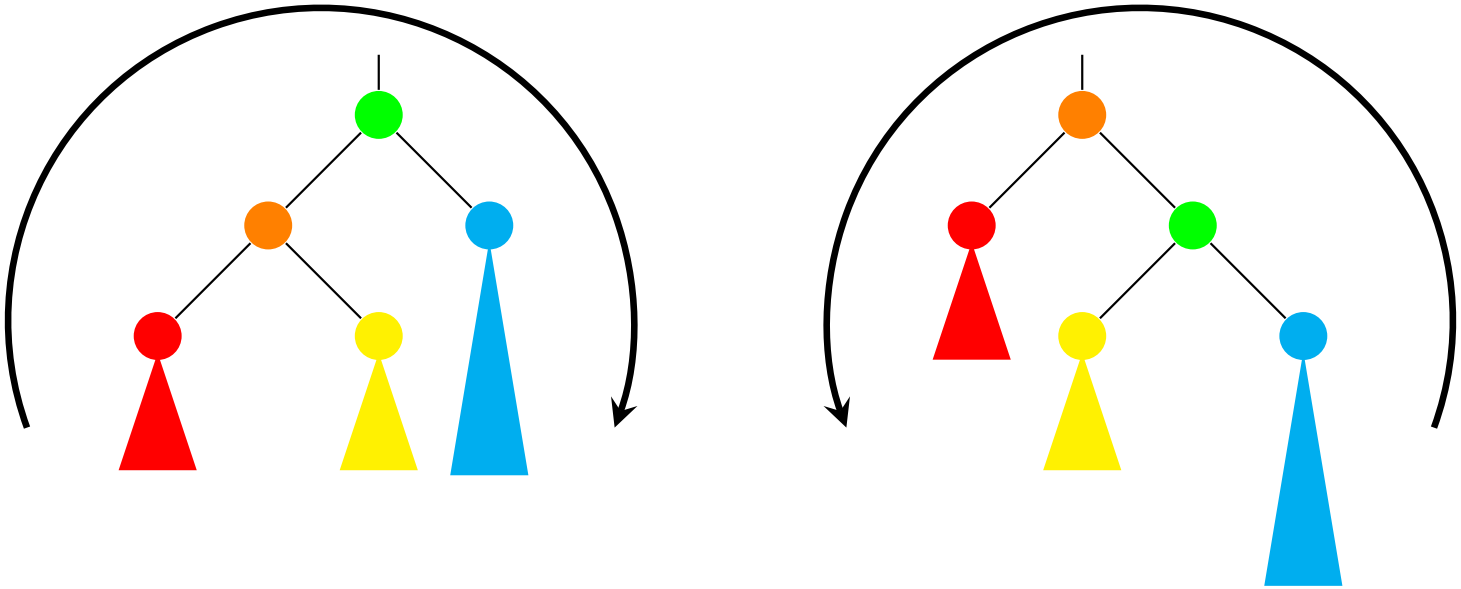
- `SEARCH(key)`: $O(h)$ – `INSERT(element)`: $O(h)$
- `DELETE(position)`: $O(h)$

Binary Search Trees: the height problem



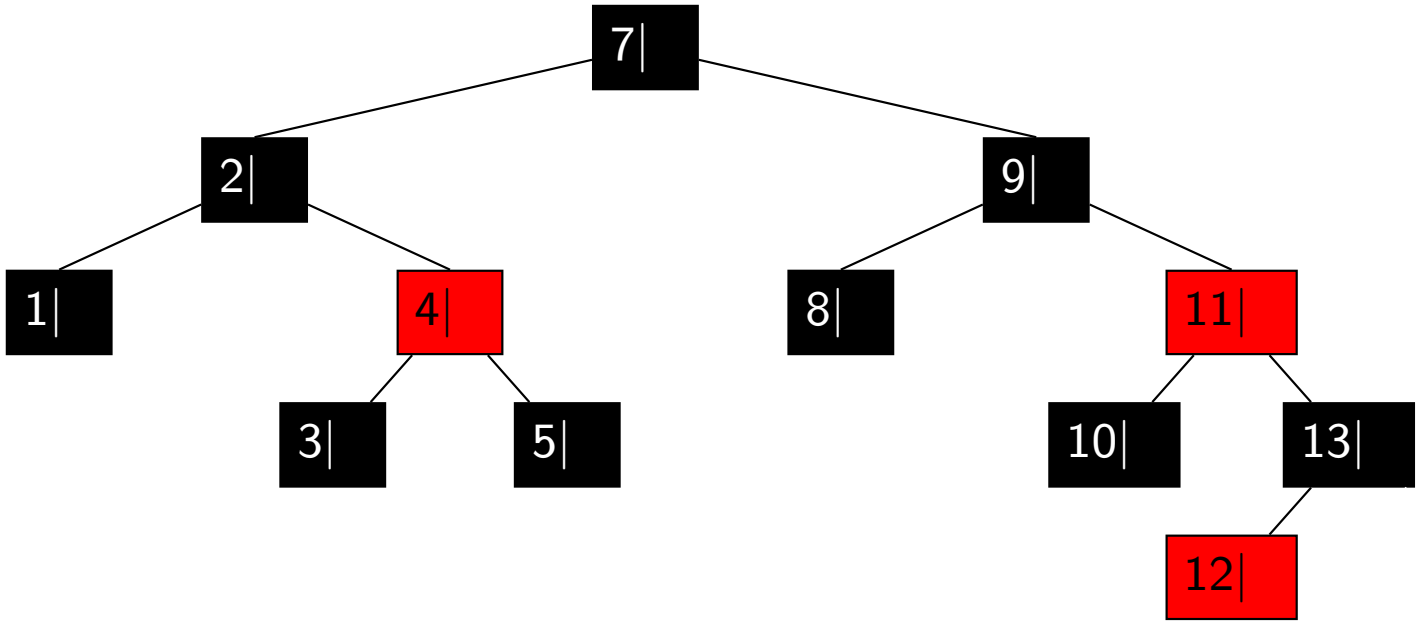
- `SEARCH(key)`, `INSERT(element)`, `DELETE(position)` are all $O(h)$
- but h may be equal to n.

Binary Search Trees: Rotations



Running time of a rotation: $\Theta(1)$.
Rotations may change the height!

Red-Black Trees: the structure



A RBT is a binary search tree with each node colored red or black such that

- The root is black.
- Every leaf (NIL) is black.
- The children of a red node are black.
- From any node, all paths to the leaves have the same number of black nodes.

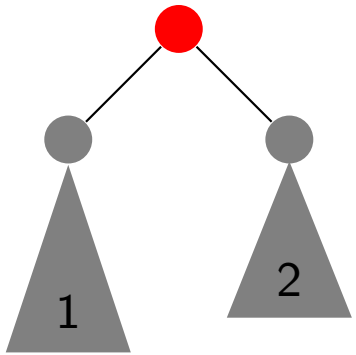
Remark: the subtree rooted at any node of a RBT is a RBT, except for the color of the root.

Red-Black Trees: Log-bounded Height

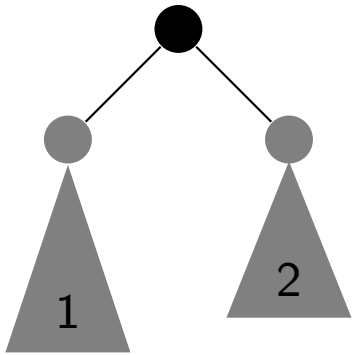
$$b \leq \log_2(n+1) \begin{cases} n: \text{ number of nodes} \\ b: \text{ number of black nodes on paths from root to leaves} \end{cases}$$

Proof by *structural induction*:

- If the tree is empty, $b = 0$ and $n = 0$, thus $b \leqslant log_2(n+1)$.
- If the property is true for the children, then it is true for the tree:



$$\begin{aligned} b &= b_1 = b_2 \\ &\leq \log_2(n_1+1) \\ &< \log_2(n+1) \end{aligned}$$



$$\begin{aligned} b &= b_1 + 1 = b_2 + 1 \\ &\leq \log_2(\min(n_1, n_2) + 1) + log_2 2 \\ &\leq \log_2(2\min(n_1, n_2) + 2) \\ &\leq \log_2(n+1) \end{aligned}$$

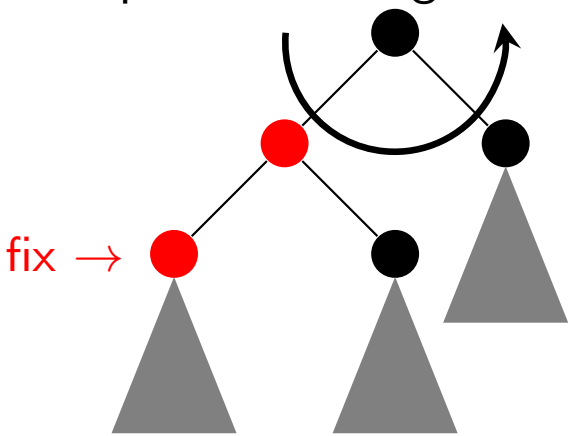
Since the children of a red node are black, $h \leq 2 \times b$ and thus

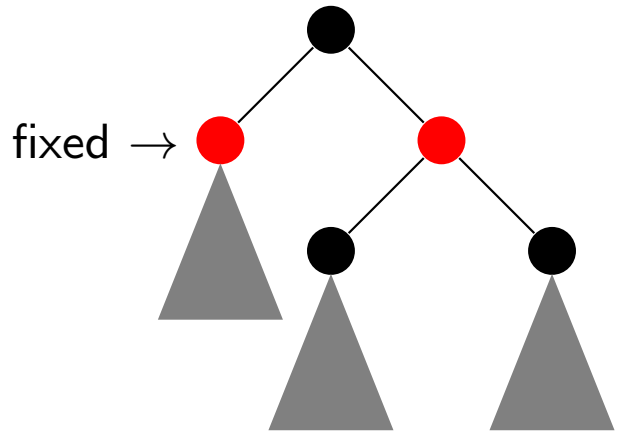
$$h \leq 2\log_2(n+1)$$

Red-Black Trees: INSERT

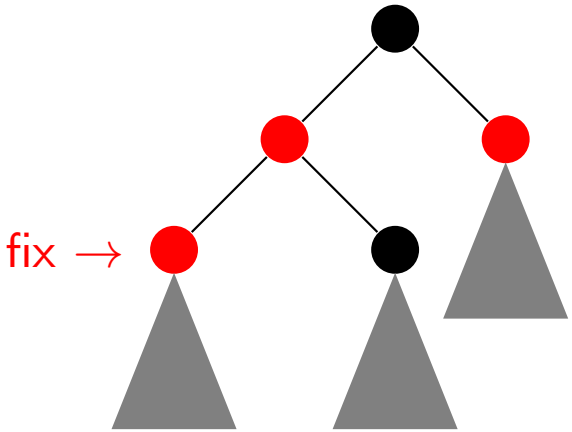
Use the INSERT algorithm for Binary Search Trees and colour the new node red.
If its parent is black the tree is still a RBT, otherwise fix it up.

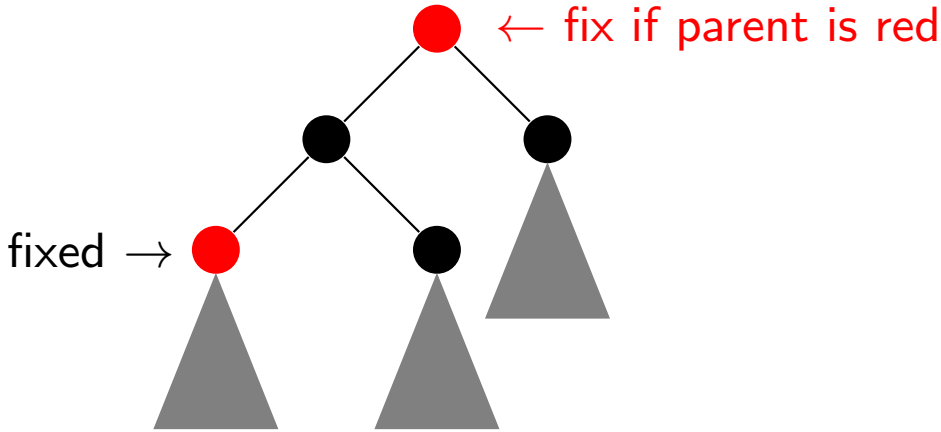
- If the parent's sibling is black, rotate and recolour.





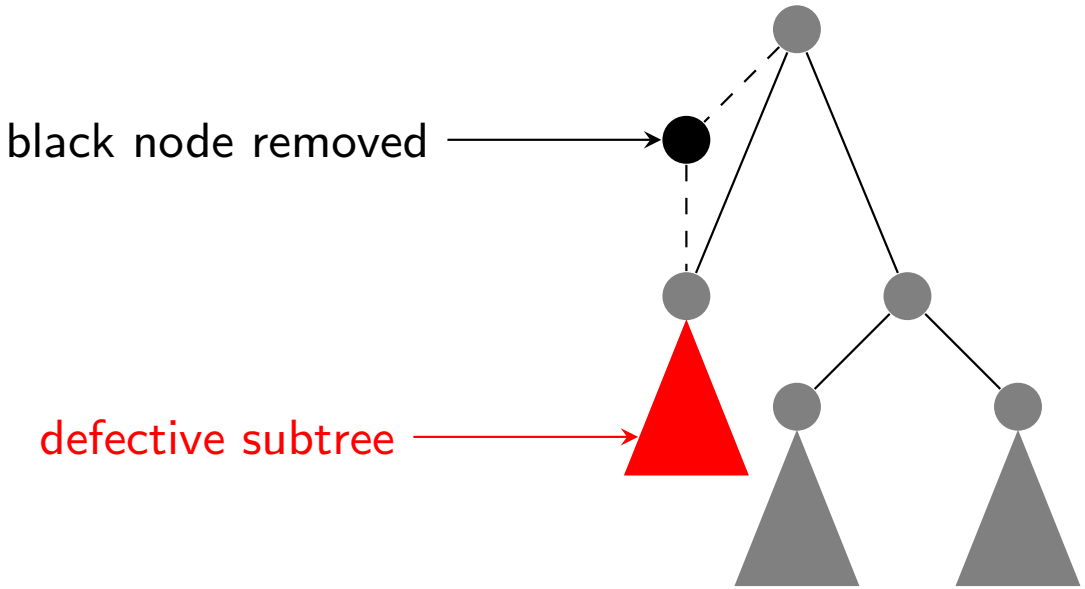
- Otherwise, recolour and recursively fix up if necessary (at most h steps).





Red-Black Trees: DELETE

Use the DELETE algorithm for Binary Search Trees (in time $O(h)$).
If a red or childless node was removed, the tree is still a RBT.
Otherwise, the removal created a defective subtree:



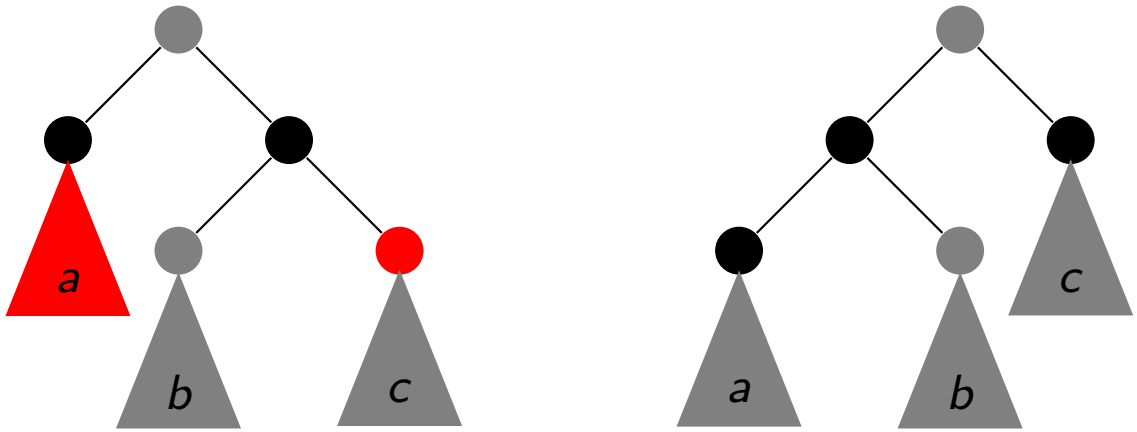
Fix up the defective subtree:

- If its root is red, colour it black.

Red-Black Trees: `DELETE` (continued)

Fix up the defective subtree (continued):

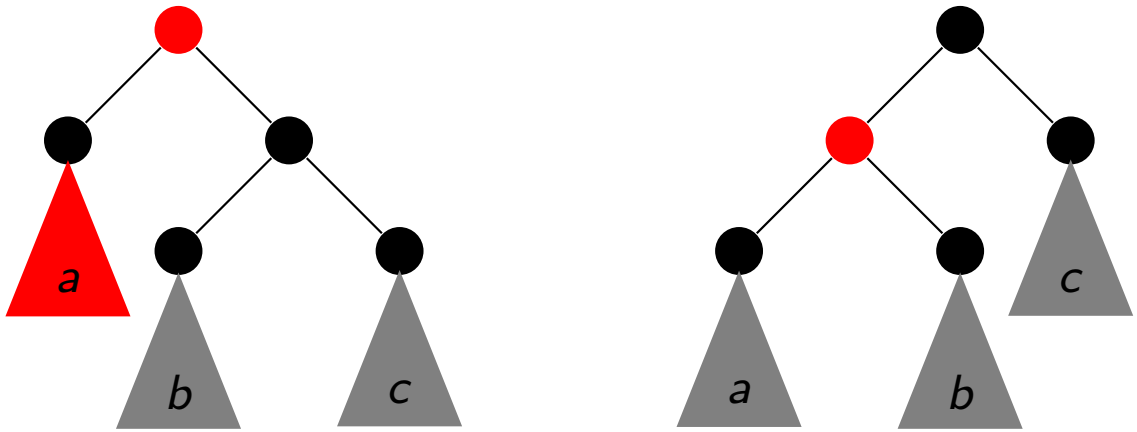
- If its root has a black sibling with a red child, rotate and recolour.



Red-Black Trees: `DELETE` (continued)

Fix up the defective subtree (continued):

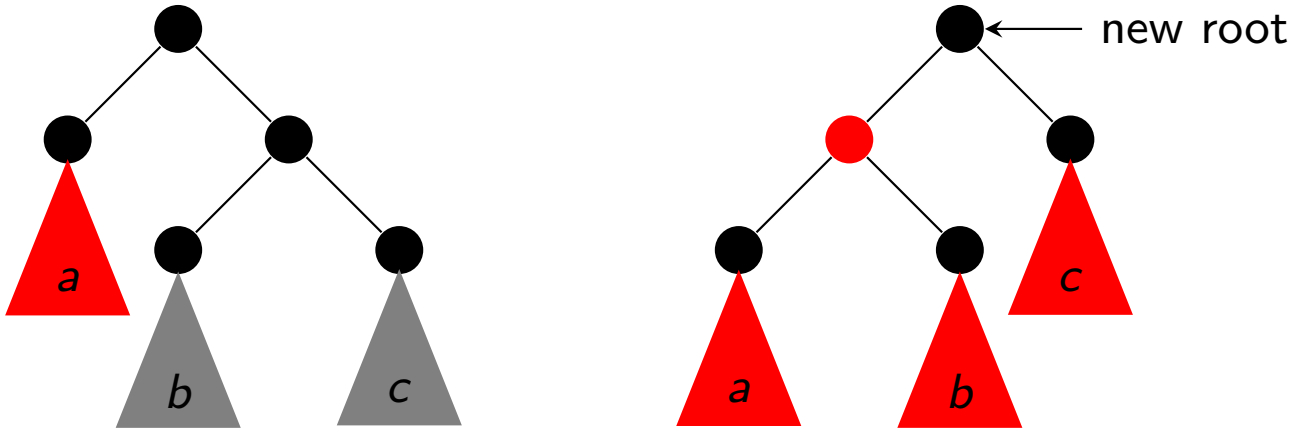
- If its root has a black sibling with black children, and a red parent, rotate and recolour.



Red-Black Trees: DELETE (continued)

Fix up the defective subtree (continued):

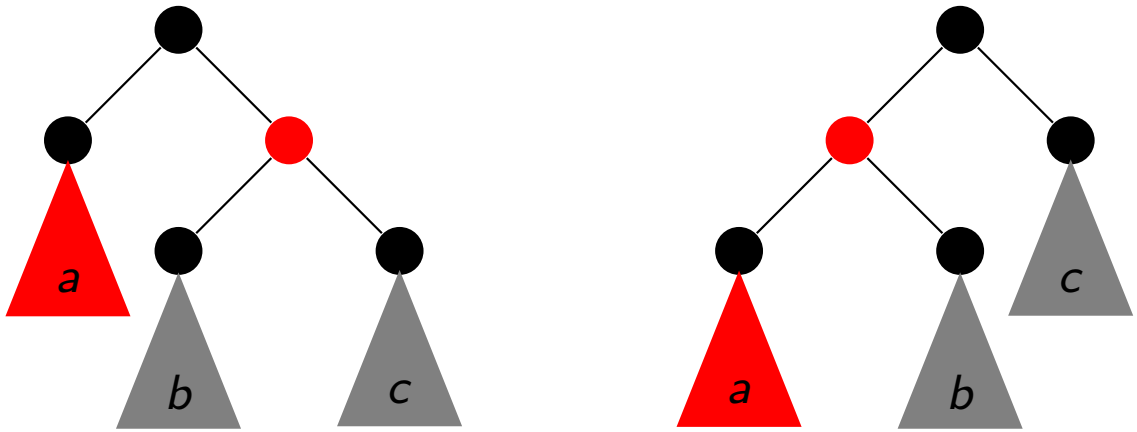
- If the root has a black sibling with black children, and a black parent: rotate, recolour and recursively fix up.



Red-Black Trees: `DELETE` (continued)

Fix up the defective subtree (continued):

- If the root has a red sibling, rotate, recolour and recursively fix up.



The root of the defective tree will never move above the red parent.

Thus the total number of recursive fixes is at most $2 \times h$, and finally `DELETE(position)` runs in time $O(\log_2 n)$.