

# 04 – Treaps



# The dictionary problem

---

**Given:** Universe  $(U, <)$  of keys with a total order

**Goal:** Maintain set  $S \subseteq U$  under the following operations

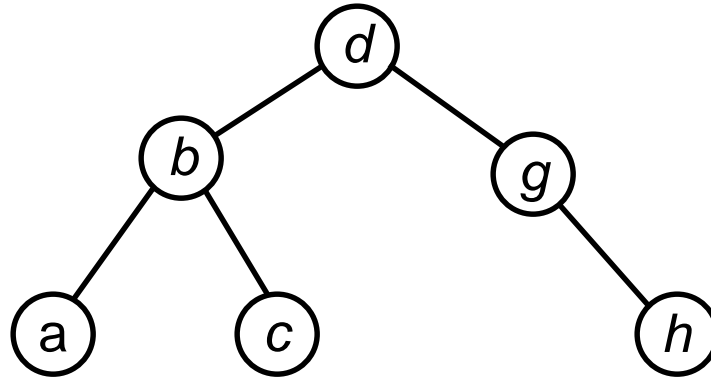
- **Search** $(x, S)$ : Is  $x \in S$ ?
- **Insert** $(x, S)$ : Insert  $x$  into  $S$  if not already in  $S$ .
- **Delete** $(x, S)$ : Delete  $x$  from  $S$ .

# Extended set of operations

- **Minimum( $S$ ):** Return smallest key.
- **Maximum( $S$ ):** Return largest key.
- **List( $S$ ):** Output elements of  $S$  in increasing order of key.
- **Union( $S_1, S_2$ ):** Merge  $S_1$  and  $S_2$ .  
Condition:  $\forall x_1 \in S_1, x_2 \in S_2: x_1 < x_2$
- **Split( $S, x, S_1, S_2$ ):** Split  $S$  into  $S_1$  and  $S_2$ .  
 $\forall x_1 \in S_1, x_2 \in S_2: x_1 \leq x$  and  $x < x_2$

# Known solutions

- **Binary search trees**



Drawback: Sequence of insertions may lead to a linear list  $a, b, c, d, e, f$

- **Height balanced trees:** AVL trees, (a,b)-trees

Drawback: Complex algorithms or high memory requirements.

If  $n$  elements are inserted in random order into a binary search tree, the expected depth is  $1.39 \log n$ .

**Idea:** Each element  $x$  is assigned a priority chosen uniformly at random

$$\text{prio}(x) \in \mathbb{R}$$

The goal is to establish the following property.

(\*) The search tree has the structure that would result if elements were inserted in the order of their priorities.

# Treaps (Tree + Heap)

**Definition:** A treap is a binary tree.

Each node contains one element  $x$  with  $\text{key}(x) \in U$  and  $\text{prio}(x) \in \mathbb{R}$ .

The following properties hold.

- **Search tree property**

For each element  $x$ :

- elements  $y$  in the left subtree of  $x$  satisfy:  $\text{key}(y) < \text{key}(x)$
- elements  $y$  in the right subtree of  $x$  satisfy :  $\text{key}(y) > \text{key}(x)$

- **Heap property**

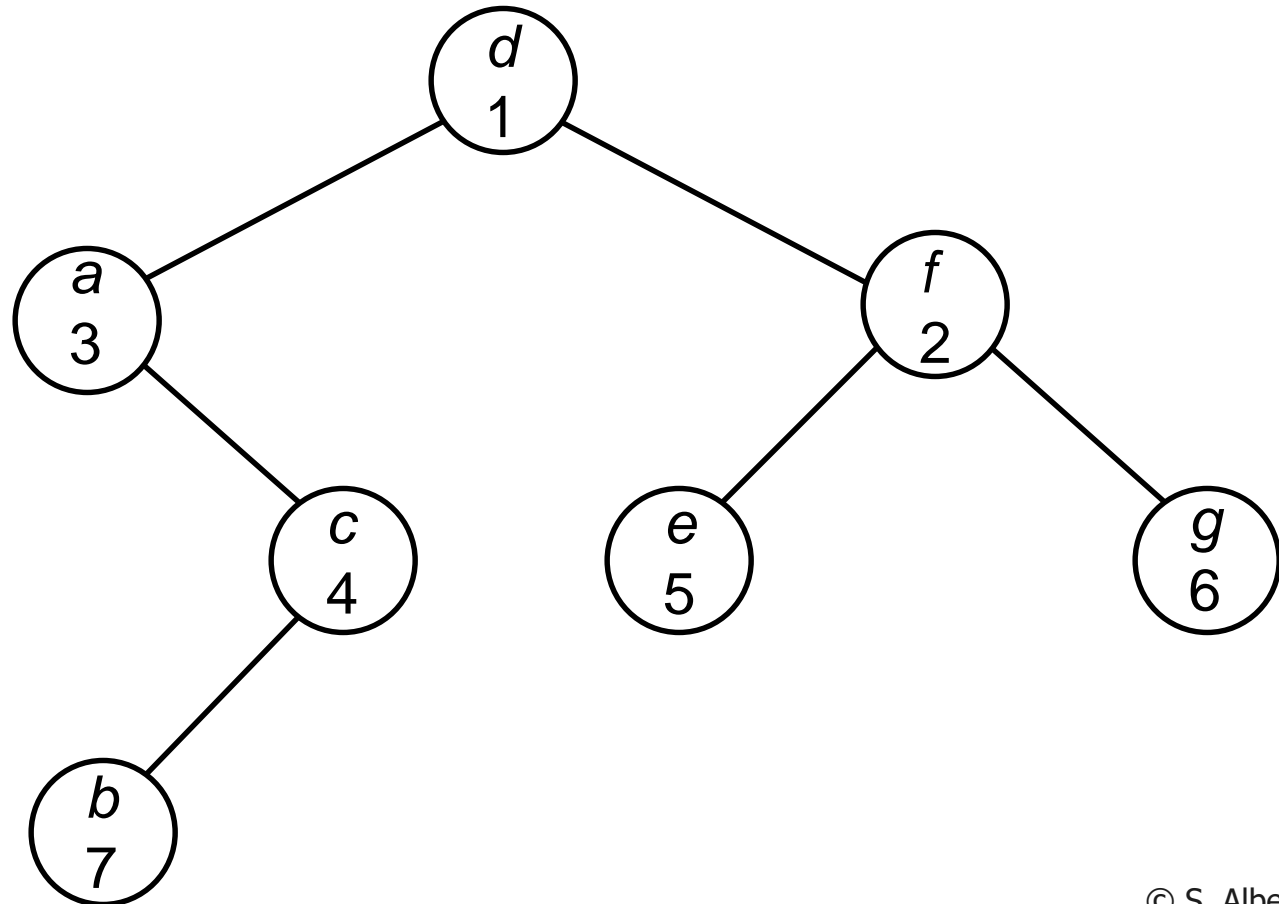
For all elements  $x, y$ :

If  $y$  is a child of  $x$ , then  $\text{prio}(y) > \text{prio}(x)$ .

All priorities are pairwise distinct.

# Example

key	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>	<i>f</i>	<i>g</i>
priority	3	7	4	1	5	2	6



# Treap uniqueness

**Lemma:** For elements  $x_1, \dots, x_n$  with  $\text{key}(x_i)$  and  $\text{prio}(x_i)$ , there exists a unique treap. It satisfies property (\*).

**Proof:**

$n=1$ : obvious

Suppose that lemma holds for element sets up to cardinality  $n-1$ .

$n-1 \Rightarrow n$ : The element  $x_i$  with smallest priority among  $x_1, \dots, x_n$  must be in the root.

Elements  $x_j$  with  $\text{key}(x_j) < \text{key}(x_i)$  are in the left subtree of  $x_i$ .

Elements  $x_j$  with  $\text{key}(x_j) > \text{key}(x_i)$  are in the right subtree of  $x_i$ .

By induction hypothesis there exists a unique treap for the elements in the left/right subtrees of  $x_i$ .

Hence there exists a unique treap for  $x_1, \dots, x_n$ .



# Treap uniqueness

---

If the elements are inserted in order of increasing priority, then element  $x_i$  with smallest priority is inserted first and resides in the root.

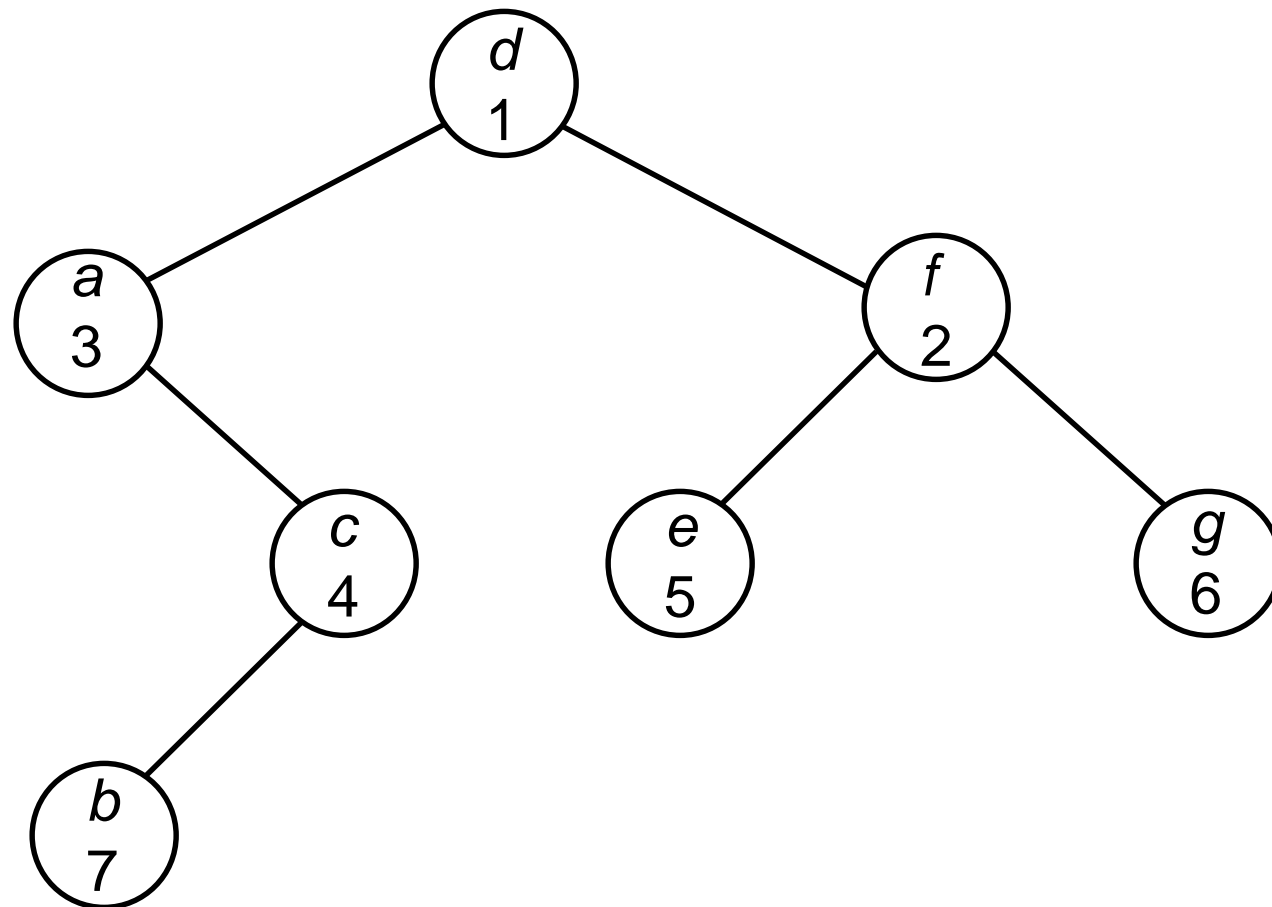
Elements  $x_j$  with  $\text{key}(x_j) < \text{key}(x_i)$  are in the left subtree of  $x_i$ .

Elements  $x_j$  with  $\text{key}(x_j) > \text{key}(x_i)$  are in the right subtree of  $x_i$ .

By induction hypothesis the treaps in the left/right subtrees of  $x_i$  have the same structure as if the respective elements were inserted in order of increasing priorities.

Hence property (\*) holds.

# Search for an element



# Search for element with key $k$

```
1   $v := \text{root};$ 
2  while  $v \neq \text{nil}$  do
3      case  $\text{key}(v) = k$  : stop; “element found” (successful search)
4           $\text{key}(v) < k$  :  $v := \text{RightChild}(v);$ 
5           $\text{key}(v) > k$  :  $v := \text{LeftChild}(v);$ 
6      endcase;
7  endwhile;
8  “element not found” (unsuccessful search)
```

Running time:  $O(\# \text{ elements on the search path})$

# Analysis of the search path

Elements  $x_1, \dots, x_n$        $x_i$  has  $i$ -th smallest key

Let  $M$  be a subset of the elements.

$P_{\min}(M)$  = element in  $M$  with lowest priority

## Lemma:

a) Let  $i < m$ .     $x_i$  is ancestor of  $x_m$     iff     $P_{\min}(\{x_i, \dots, x_m\}) = x_i$

b) Let  $m < i$ .     $x_i$  is ancestor of  $x_m$     iff     $P_{\min}(\{x_m, \dots, x_i\}) = x_i$

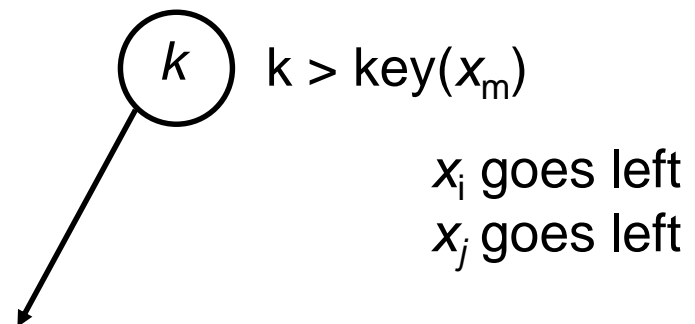
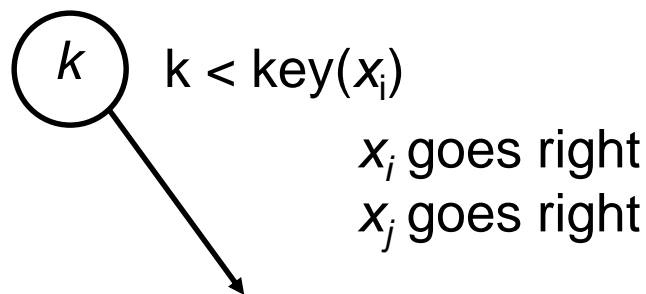
# Analysis of the search path

**Proof:** a) Use (\*). Elements are inserted in order of increasing priorities.

“ $\Leftarrow$ ”  $P_{\min}(\{x_i, \dots, x_m\}) = x_i \Rightarrow x_i$  is inserted first among  $\{x_i, \dots, x_m\}$ .

When  $x_i$  is inserted, the tree contains only keys  $k$  with  
 $k < \text{key}(x_i)$  or  $k > \text{key}(x_m)$

When  $x_j$ , with  $i < j \leq m$ , is inserted, it traverses the same search path as  $x_i$ . Hence  $x_j$  becomes a descendent of  $x_i$ .



# Analysis of the search path

**Proof:** a) (Let  $i < m$ .  $x_i$  is ancestor of  $x_m$  iff  $P_{\min}(\{x_i, \dots, x_m\}) = x_i$ )

“ $\Rightarrow$ ” Let  $x_j = P_{\min}(\{x_i, \dots, x_m\})$ . Show:  $x_i = x_j$

Suppose:  $x_i \neq x_j$

When  $x_j$  is inserted, the tree contains only keys  $k$  with

$k < \text{key}(x_i)$  or  $k > \text{key}(x_m)$

All elements of  $\{x_i, \dots, x_m\} \setminus \{x_j\}$  traverse the same search path as  $x_j$ :

Node with key  $k < \text{key}(x_i)$ : All elements from  $\{x_i, \dots, x_m\}$  turn right.

Node with key  $k > \text{key}(x_m)$ : All elements from  $\{x_i, \dots, x_m\}$  turn left.

Hence all elements of  $\{x_i, \dots, x_m\} \setminus \{x_j\}$  become descendants of  $x_j$ .

Case 1:  $x_j = x_m$   $x_i$  is descendent of  $x_m$  Contradiction!

Case 2:  $x_j \neq x_m$   $x_i$  and  $x_m$  are in different subtrees of  $x_j$  Contradiction!

Part b) can be shown analogously.

# Analysis of the 'Search' operation

Let  $T$  be a treap with elements  $x_1, \dots, x_n$   $x_i$  has  $i$ -th smallest key

$n$ -th Harmonic number:

$$H_n = \sum_{k=1}^n 1/k$$

## Lemma:

1. **Successful search:** The expected number of nodes on the path to  $x_m$  is  $H_m + H_{n-m+1} - 1$ .
2. **Unsuccessful search :** Let  $m$  be the number of keys that are smaller than the search key  $k$ . The expected number of nodes on the search path is  $H_m + H_{n-m}$ .

# Analysis of the 'Search' operation

## Proof: Part 1

$$X_{m,i} = \begin{cases} 1 & x_i \text{ is ancestor of } x_m \\ 0 & \text{otherwise} \end{cases}$$

$X_m = \#$  nodes on the path from the root to  $x_m$  (incl.  $x_m$ )

$$X_m = 1 + \sum_{i < m} X_{m,i} + \sum_{i > m} X_{m,i}$$

$$E[X_m] = 1 + E\left[\sum_{i < m} X_{m,i}\right] + E\left[\sum_{i > m} X_{m,i}\right]$$



# Analysis of the 'Search' operation

---

$i < m$ :

$$E[X_{m,i}] = \text{Prob}[x_i \text{ is ancestor of } x_m] = 1/(m - i + 1)$$

All elements in  $\{x_i, \dots, x_m\}$  have the same probability of being the one with the smallest priority.

$$\text{Prob}[P_{\min}(\{x_i, \dots, x_m\}) = x_i] = 1/(m - i + 1)$$

$i > m$ :

$$E[X_{m,i}] = 1/(i - m + 1)$$

# Analysis of the 'Search' operation

$$\begin{aligned} E[X_m] &= 1 + \sum_{i < m} \frac{1}{m - i + 1} + \sum_{i > m} \frac{1}{i - m + 1} \\ &= 1 + \frac{1}{m} + \dots + \frac{1}{2} + \frac{1}{2} + \dots + \frac{1}{n - m + 1} \\ &= H_m + H_{n-m+1} - 1 \end{aligned}$$

# Analysis of the 'Search' operation

## Part 2

$m = 0$ : Search path is the same as that for  $x_1$ . By Part 1, the expected number of nodes on the search path is  $H_1 + H_n - 1 = H_n$ .

$m = n$ : Search path is the same as that for  $x_n$ . By Part 1, the expected number of nodes on the search path is  $H_n + H_1 - 1 = H_n$ .

$0 < m < n$ :  $x_m$  is an ancestor of  $x_{m+1}$  or vice versa. When searching for  $k$ , at every

$x_i$  with  $i < m$ , the search path turns right

$x_i$  with  $i > m+1$ , the search path turns left.

Hence the **search path for  $k$**  is the same as that for  $x_m, x_{m+1}$  until **one of the two keys are hit**.

# Analysis of the 'Search' operation

If  $x_m$  is hit first, the remaining search path of  $k$  is identical to that of  $x_{m+1}$ .

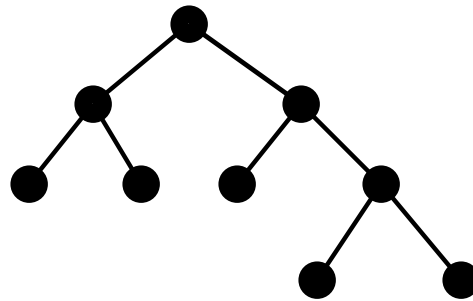
If  $x_{m+1}$  is hit first, the remaining search path of  $k$  is identical to that of  $x_m$ .



Hence the length of the search path is upper bounded by that of  $x_m$  and  $x_{m+1}$ , i.e.  $\max \{H_m + H_{n-m+1} - 1, H_{m+1} + H_{n-m} - 1\} \leq H_m + H_{n-m}$ .

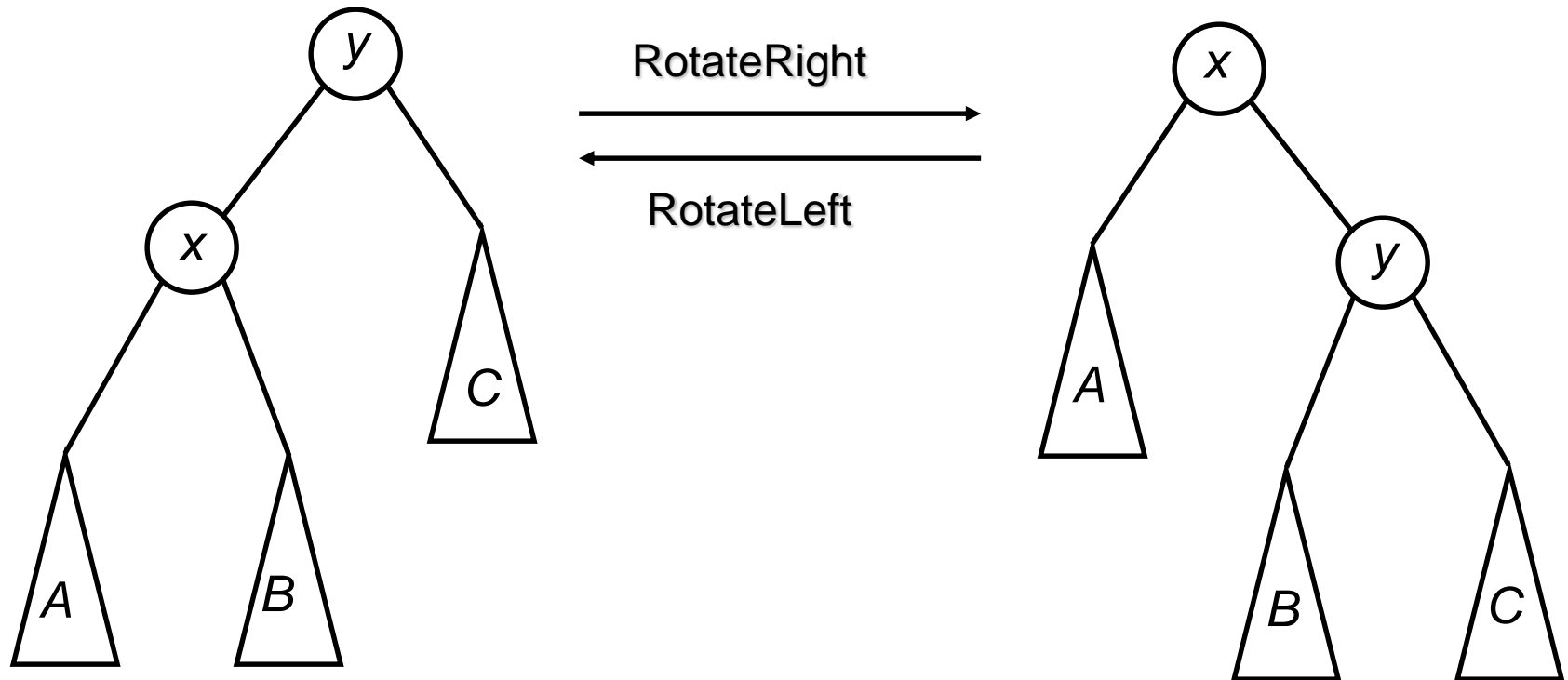
# Inserting a new element $x$

1. Choose  $\text{prio}(x)$ .
2. Search for the **position** of  $x$  in the tree.



3. Insert  $x$  as a leaf.
4. Restore the **heap property**.  
**while**  $\text{prio}(\text{parent}(x)) > \text{prio}(x)$  **do**  
    **if**  $x$  is left child **then**  $\text{RotateRight}(\text{parent}(x))$   
    **else**  $\text{RotateLeft}(\text{parent}(x));$   
**endif**  
**endwhile;**

# Rotations



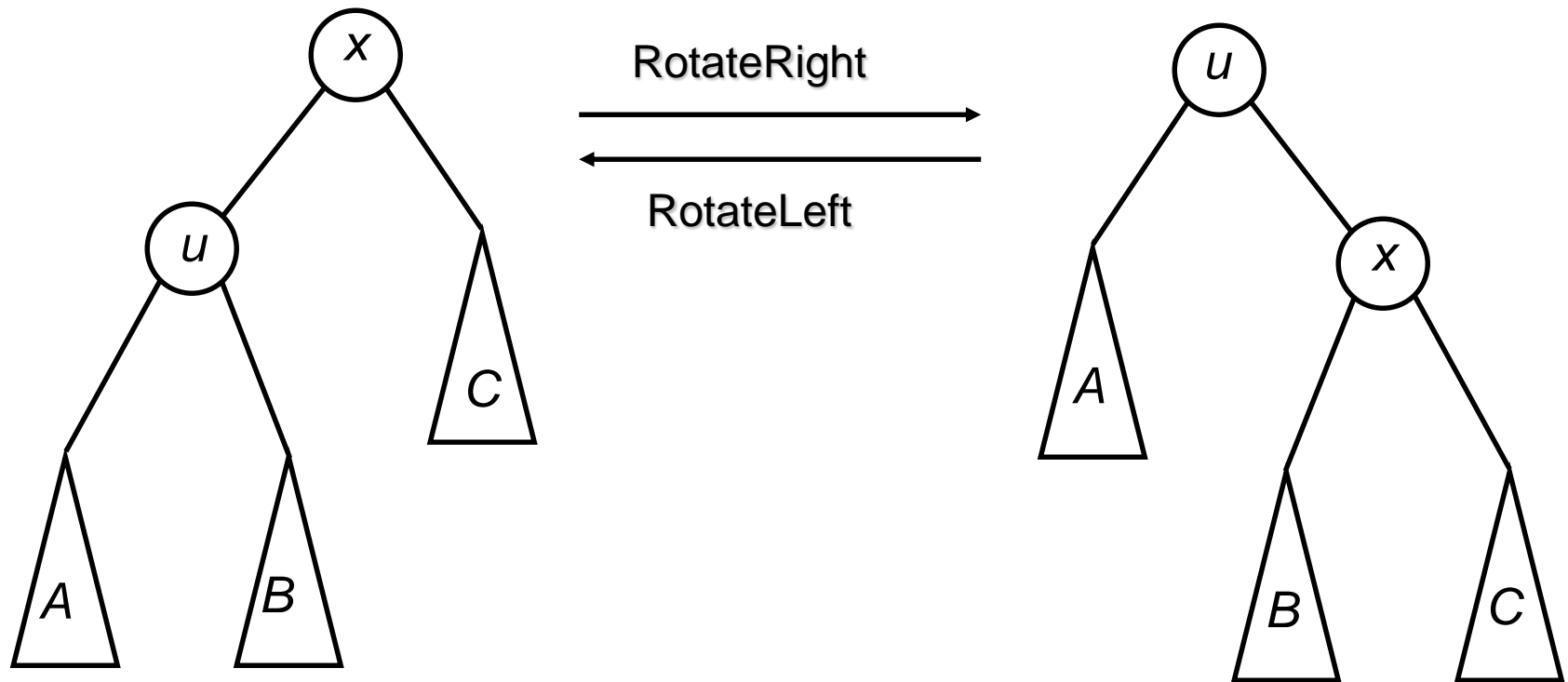
The rotations maintain the search tree property and restore the heap property.

# Deleting an element $x$

---

1. Find  $x$  in the tree.
2. **while**  $x$  is not a leaf **do**
  - $u :=$  child with smaller priority;
  - if**  $u$  is left child **then** RotateRight( $x$ )
    - else** RotateLeft( $x$ );
  - endif**;**endwhile**;
3. Delete  $x$ ;

# Rotations





# Analysis of 'Insert' and 'Delete' operations

**Lemma:** The expected running time of insert and delete operations is  $O(\log n)$ . The expected number of rotations is 2.

**Proof:** Analysis of insert (delete is the inverse operation)

# rotations = depth of  $x$  after being inserted as a leaf (1)

- depth of  $x$  after the rotations (2)

Let  $x = x_m$ .

(2) Expected depth is  $H_m + H_{n-m+1} - 1$ .

(1) Expected depth is  $H_{m-1} + H_{n-m} + 1$ .

The tree contains  $n-1$  elements,  $m-1$  of them being smaller.

$$\# \text{ rotations} = H_{m-1} + H_{n-m} + 1 - (H_m + H_{n-m+1} - 1) < 2$$

# Extended set of operations

$n$  = number of elements in treap  $T$ .

- **Minimum( $T$ ):** Return the smallest key.  $O(\log n)$
- **Maximum( $T$ ):** Return the largest key.  $O(\log n)$
- **List( $T$ ):** Output elements of  $S$  in increasing order.  $O(n)$
  
- **Union( $T_1, T_2$ ):** Merge  $T_1$  and  $T_2$ .  
Condition:  $\forall x_1 \in T_1, x_2 \in T_2: \text{key}(x_1) < \text{key}(x_2)$
- **Split( $T, k, T_1, T_2$ ):** Split  $T$  into  $T_1$  and  $T_2$ .  
 $\forall x_1 \in T_1, x_2 \in T_2: \text{key}(x_1) \leq k$  and  $k < \text{key}(x_2)$

# The 'Split' operation

$\text{Split}(T, k, T_1, T_2)$ : Split  $T$  into  $T_1$  and  $T_2$ .

$$\forall x_1 \in T_1, x_2 \in T_2: \text{key}(x_1) \leq k \text{ and } \text{key}(x_2) > k$$

W.l.o.g. key  $k$  is not in  $T$ .

Otherwise delete the element with key  $k$  and re-insert it into  $T_1$  after the split operation.

1. Generate a new element  $x$  with  $\text{key}(x)=k$  and  $\text{prio}(x) = -\infty$ .
2. Insert  $x$  into  $T$ .
3. Delete the new root. The left subtree is  $T_1$ , the right subtree is  $T_2$ .

# The 'Union' operation

$\text{Union}(T_1, T_2)$ : Merge  $T_1$  and  $T_2$ .

Condition:  $\forall x_1 \in T_1, x_2 \in T_2: \text{key}(x_1) < \text{key}(x_2)$

1. Determine key  $k$  with  $\text{key}(x_1) < k < \text{key}(x_2)$   
for all  $x_1 \in T_1$  and  $x_2 \in T_2$ .
2. Generate element  $x$  with  $\text{key}(x)=k$  and  $\text{prio}(x) = -\infty$ .
3. Generate treap  $T$  with root  $x$ , left subtree  $T_1$  and  
right subtree  $T_2$ .
4. Delete  $x$  from  $T$ .

**Lemma:** The expected running time of the operations **Union** and **Split** is  $O(\log n)$ .

# Implementation

---

Priorities from  $[0,1)$

Priorities are used only when two elements are compared to find out which of them has the higher priority.

In case of equality, extend both priorities by bits chosen uniformly at random until two corresponding bits differ.

$$p_1 = 0.010111001$$

$$p_2 = 0.010111001$$

$$p_1 = 0.010111001011$$

$$p_2 = 0.010111001010$$