Algorithms Theory

04 - Treaps

Dr. Alexander Souza

The dictionary problem

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

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

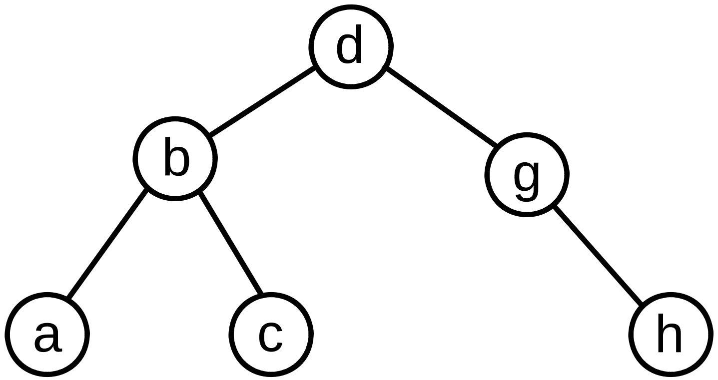
- Search(x,S): Is $x \in S$?
- Insert(x,S): Insert x into S if not already in S.
- Delete(x,S): Delete x from S.

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.

Approach for randomized search trees

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

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

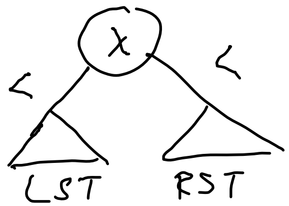
The goal is to establish the following property.

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

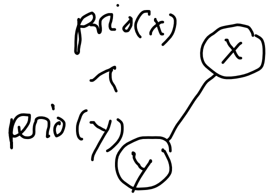
Treaps (Tree + Heap)

Definition: A treap is a binary tree.
Each node contains one element x with key(x) $\in U$ and prio(x) $\in R$.
The following properties hold.

- Search tree property
  For each element x:
  - elements y in the left subtree of x satisfy: key(y) < key(x)
  - elements y in the right subtree of x satisfy : key(y) > key(x)

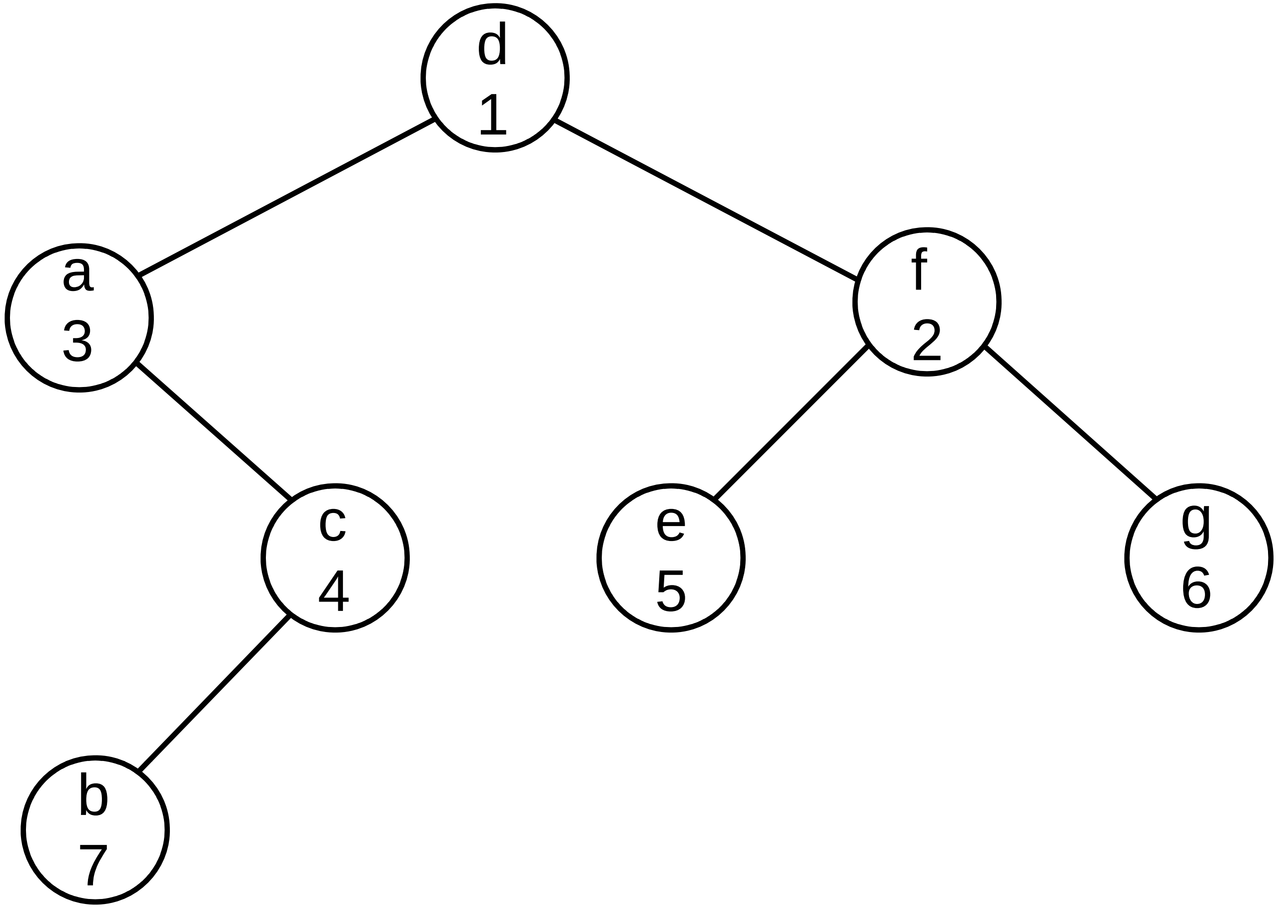


- Heap property
  For all elements x,y:
  If y is a child of x, then prio(y) > prio(x).
  All priorities are pairwise distinct.



Example

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



Treap uniqueness

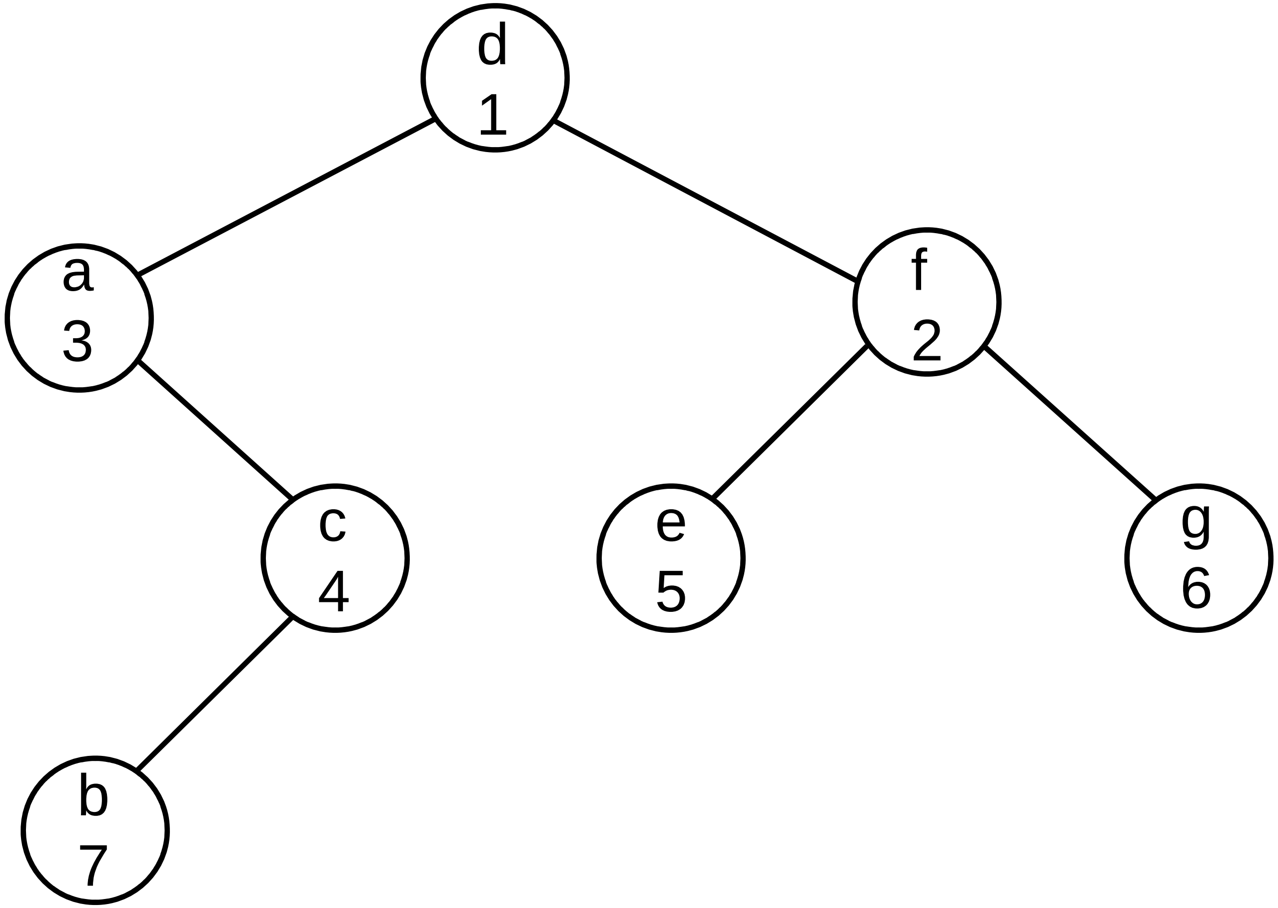
Lemma: For elements $x_1, \ldots , x_n$ with key(x_i) and prio(x_i), there exists a unique treap. It satisfies property (*).

Proof:

n=1: ok

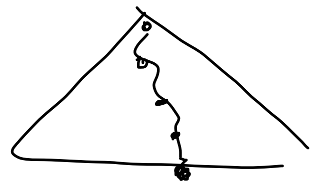
n>1:

Search for an element



Search for element with key *k*

```
v := root;
while v ≠ nil do
   case key(v) =  k : stop;   "element found" (successful search)
              key(v) <  k : v:= RightChild(v);
              key(v) >  k : v:= LeftChild(v);
   endcase;
endwhile;
"element not found" (unsuccessful search)
```



Running time: O(# elements on the search path)

Analysis of the search path

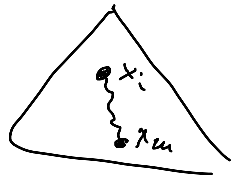
$key(x_1) < key(x_2) < \ldots < key(x_n)$

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

Let M be a subset of the elements.

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

$x_1, \ldots, x_i, \ldots, x_m, \ldots, x_n$



Lemma:

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

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

Analysis of the search path

$\text{key}(x_1) < \text{key}(x_2) < \ldots < \text{key}(x_m)$

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

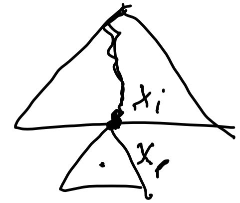
"$\Leftarrow$" $P_{min}(\{x_i,\ldots,x_m\}) = x_i \quad \Rightarrow \quad x_i$ is inserted first among $\{x_i,\ldots,x_m\}$.

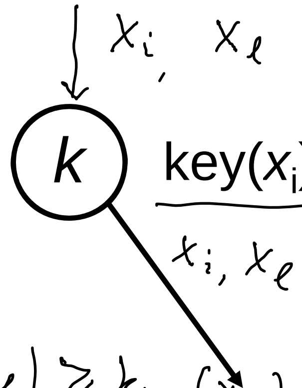
When x_i is inserted, the tree contains only keys k with

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

We show: When an element $x_\ell \in \{x_i,\ldots,x_m\}$ is inserted, it traverses the same search path as x_i.

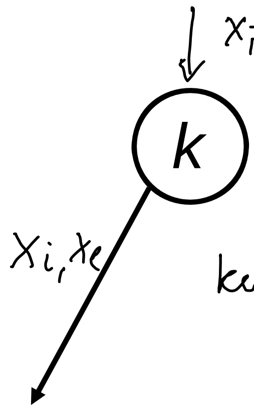
$\Rightarrow$ x_i ancestor of x_ℓ $\Rightarrow$ x_i ancestor of x_m ✓





key(x_i) > k

$\text{key}(x_\ell) \geq \text{key}(x_i) > k$



key(x_i) < k

$\text{key}(x_\ell) \leq \text{key}(x_m) < k$

Analysis of the search path

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

"$\Rightarrow$" Let $x_j = P_{min}(\{x_i,\ldots,x_m\})$. Show: $x_i = x_j$

Suppose: $x_i \neq x_j$

Consider the search path when x_j is inserted.

As before any $x_\ell \in \{x_i,\ldots,x_m\}$ traverses the same search path as x_j

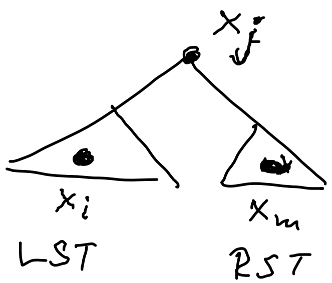
x_j is ancestor of x_ℓ

Case 1: $x_j = x_m$ $\Rightarrow$ x_m is ancestor of x_ℓ $\Rightarrow$ x_m is ancestor of x_i ↯

Case 2: $x_j \neq x_m$

key $(x_i) <$ key (x_j)

key $(x_m) >$ key (x_j)



Part b) follows analogously. $\Rightarrow$ x_i is not ancestor of x_m ↯

Analysis of the 'Search' operation

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

n-th Harmonic number: $$H_n = \sum_{k=1}^{n} 1/k \cong \ln n$$

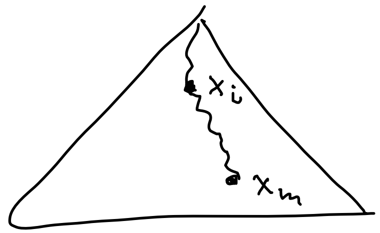
Lemma: ~~Search for key m~~

1. Successful search: The expected number of nodes on the path to x_m is $H_m + H_{n-m+1} - 1$. $= O(\log n)$ $\quad m \le n$

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 $H_m + H_{n-m}$. $= O(\log n)$ $\quad m \le n$

Analysis of the 'Search' operation

Proof: Part 1 Successful Search



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

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

$$\| \quad \sum_{i=1}^{n} X_{m,i} = X_{m,m} + \sum_{i<m} X_{m,i} + \sum_{i>m} X_{m,i}$$

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

linearity of expectation

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

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

Analysis of the 'Search' operation

$Pr[\ x_i = P_{min}(\{x_i, \ldots, x_m\})\ ] = \frac{1}{|\{x_i, \ldots, x_m\}|} = \frac{1}{m-i+1}$

i < *m* :

$\parallel$ Lemma

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

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

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

i > *m* :

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

Analysis of the 'Search' operation

$$\underline{E[X_m]} = 1 + \underbrace{\sum_{i<m} \frac{1}{m-i+1}}_{\frac{1}{m} + \frac{1}{m-1} + \dots + \frac{1}{2}} + \underbrace{\sum_{i>m} \frac{1}{i-m+1}}_{\frac{1}{2} + \dots \frac{1}{n-m+1}}$$

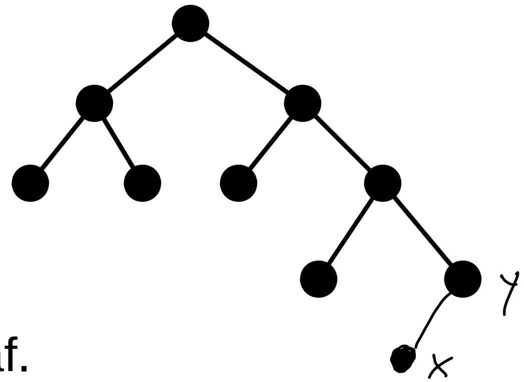
$$= 1 + \underbrace{\frac{1}{m} + \dots + \frac{1}{2}}_{H_m} + \underbrace{\frac{1}{2} + \dots + \frac{1}{n-m+1}}_{H_{n-m+1} - 1}$$

$$= H_m + H_{n-m+1} - 1$$

Part 2 follows analogously

Inserting a new element *x*

1. Choose prio(x).
2. Search for the position of *x* in the tree.



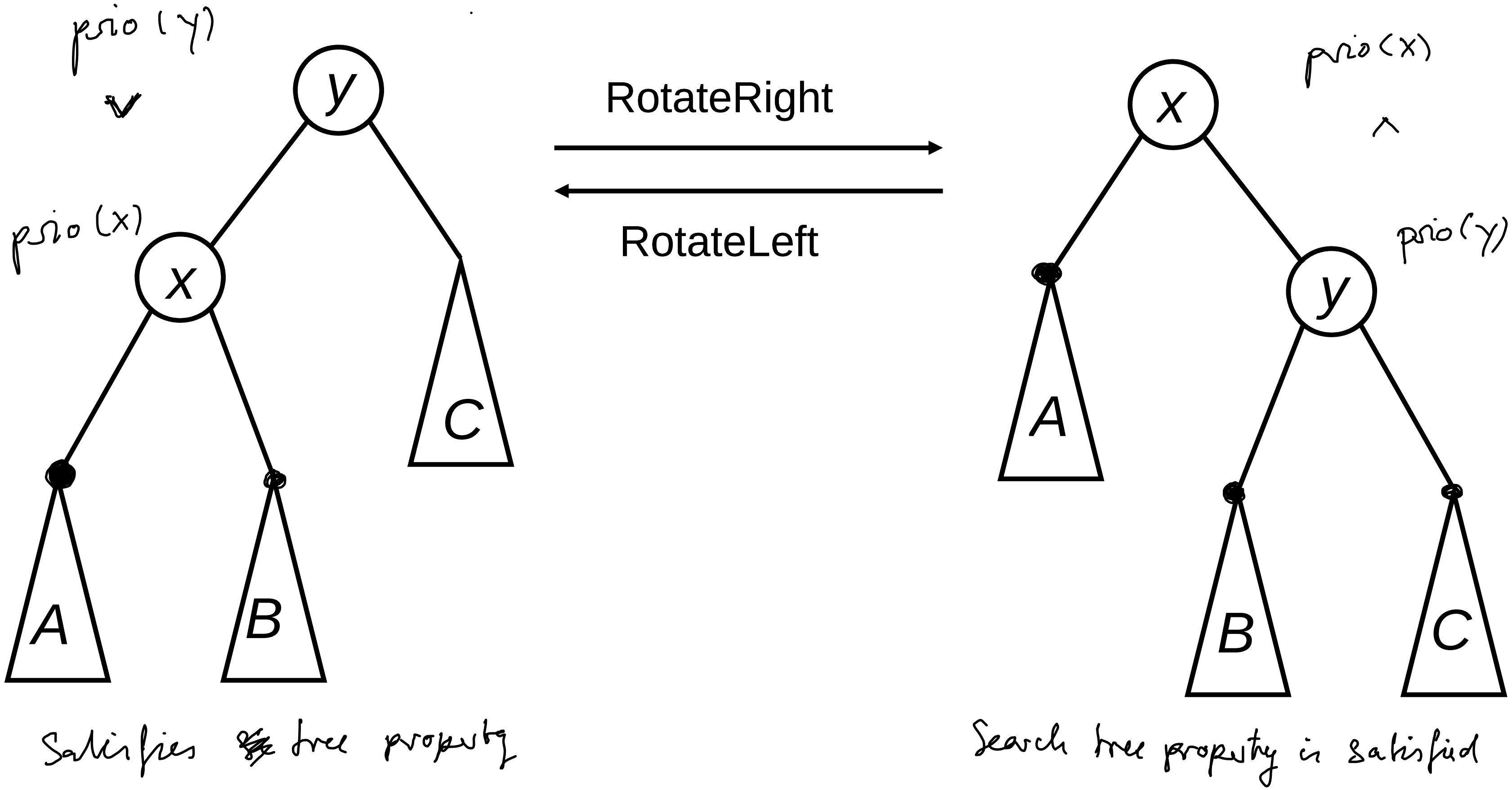
prio(y)
>
prio(x)

Heap property could be violated

3. Insert *x* as a leaf.
4. Restore the heap property.

while prio(parent(*x*)) > prio(*x*) **do**
  if *x* is left child **then** RotateRight(parent(*x*))
        else RotateLeft(parent(*x*));
  endif
endwhile;

Rotations



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