The Word RAM Model, Asymptotic Notation and the Treap

Junhao Gan Tony Wirth

School of Computing and Information Systems
The University of Melbourne

February 28, 2022

Acknowledgements

Some of the content in these slides are inspired by the following materials:

- The Word RAM Model by Yufei Tao:
  http://www.cse.cuhk.edu.hk/~taoyf/course/comp3506/lec/ram.pdf
- The Asymptotic Notation by Yufei Tao:
  http://www.cse.cuhk.edu.hk/~taoyf/course/comp3506/lec/asymptotic.pdf
- The Treap by Jeff Erickson:
  http://jeffe.cs.illinois.edu/teaching/algorithms/notes/03-treaps.pdf

Computational Model

About Theoretical Computer Science

Theoretical computer science is arguably a branch of mathematics.

In Theoretical Computer Science

Everything, e.g., concept, term, claim, symbol, notation and etc., is abstract and mathematically rigorous.

Algorithm designs are *independent* to *any* specific programming language.

These all hold in this subject.

What Distinguishes Theoretical Computer Science?

Unlike maths and physics, which are to discover laws to explain how the world operates, theoretical computer science:

- defines the laws of the world (i.e., how a machine operates) in the first place, and
- studies to design "*efficient*" solutions (i.e., the algorithms) to the given problems by manipulating the laws.

To start the study of theoretical computer science, we need to define a **computational model** first, which:

- models the machine and specifies what operations it can perform;
- defines how to measure the efficiency of a solution to a problem.

There are various computational models. Under different computational models, the philosophy in designing algorithms can be vastly different.

The Word Random Access Machine (RAM) Model

In the Word RAM model, a machine has a **memory** and a **CPU**.

Memory

The memory is an *infinite* sequence of **cells**:

- each cell can store a **word** in w bits;
- each cell is identified by an **unique address**, where the addresses start from 0 for the first cell, 1 for the second, . . ., and so on.

CPU

The CPU has a fixed number of **registers**, each of which can store a word in w bits (i.e., same as a memory cell).

Atomic Operations of the CPU

There are four atomic operations that the CPU can perform:

- **Register (Re-)Initialization**: Set the content of a register to a fixed value, or to the content of another register.
- **Arithmetic Operations**: Take two integers a and b stored in two registers, calculate the following and then store the result in a register:
  - $a + b$, $a - b$, $a \cdot b$, and integer division a/b. For example, $10/2 = 5$, $3/2 = 1$ and $4/6 = 0$.
- **Comparison or Branching**: Compare two integers a and b stored in two registers and decide which of the following is true:
  - $a < b$, $a = b$, and $a > b$.
- **Memory Access**: Take the memory address p stored in a register, then perform one of the following:
  - Read the content stored in the memory cell at address p into a specified register;
  - Write the content stored in a specified register to the memory cell at address p.

Algorithm, Running time Cost, and Space Consumption

Every **algorithm** is a sequence of the above four atomic operations.

The **running time cost** of an algorithm is measured by the number of operations in the sequence.

The **space consumption** of an algorithm is measured by the footprint of memory cells it has ever touched.

Instead of the precise number of operations (rsp., memory cells used) in an algorithm, people are more interested in **how fast the number of operations (rsp., memory cells) grows** as a function of the problem size.

Such a function is called the **running time (rsp., space) complexity** of the algorithm.

Asymptotic Notation

Consider two functions, $f(n)$ and $g(n)$, of a variable n.

Big-O Notation

We say $f(n)$ **grows asymptotically no faster than** $g(n)$, if there exist two *positive constants* c_1 and c_2 such that:

$$f(n) \leq c_1 \cdot g(n) \text{ holds for all } n \geq c_2.$$

Big-O notation $O(g(n))$ is defined as a family of functions:

$$O(g(n)) = \{h(n) \mid h(n) \text{ grows asymptotically no faster than } g(n)\}.$$

Therefore, $f(n) \in O(g(n))$.

People may (inaccurately) write $f(n) = O(g(n))$ with an abuse of notation.

True or False

$$
\begin{aligned}
1000000 &\in O(1) \\
6n+7 &\in O(n) \\
6n+7 &\in O(n^2) \\
(\log_2 n)^{999999999} &\in O(n^{0.0000000000001}) \\
\log n &\in O(n^{\epsilon}) \text{ for all constant } \epsilon > 0 \\
n^{999999999999} &\in O(2^n)
\end{aligned}
$$

Big-Ω Notation

We say $f(n)$ **grows asymptotically no slower than** $g(n)$, if there exist two *positive constants* c_1 and c_2 such that:

$$f(n) \geq c_1 \cdot g(n) \text{ holds for all } n \geq c_2.$$

Big-Ω notation $\Omega(g(n))$ is defined as a family of functions:

$$\Omega(g(n)) = \{h(n) \mid h(n) \text{ grows asymptotically no slower than } g(n)\}.$$

Therefore, $f(n) \in \Omega(g(n))$, and people may write $f(n) = \Omega(g(n))$.

Examples

$$1000000 \in \Omega(1)$$
$$n^2 + 3 \in \Omega(n)$$
$$n^{0.0000000000001} \in \Omega((\log_2 n)^{999999999})$$
$$2^n \in \Omega(n^{999999999999})$$

Big-Θ Notation

Big-Θ notation $\Theta(g(n))$ is defined as:

$$\Theta(g(n)) = O(g(n)) \cap \Omega(g(n)),$$

which is the family of all functions of n that **grow asymptotically as fast as** $g(n)$.

In other words, if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$ both hold, then $f(n) \in \Theta(g(n))$.

Again, people may write $f(n) = \Theta(g(n))$.

Treap: A Randomized Binary Search Tree

Preliminaries

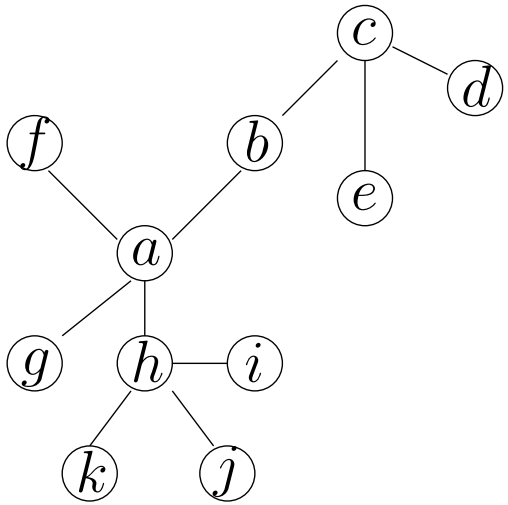
Tree

A **tree** is a connected undirected graph that contains no cycles.

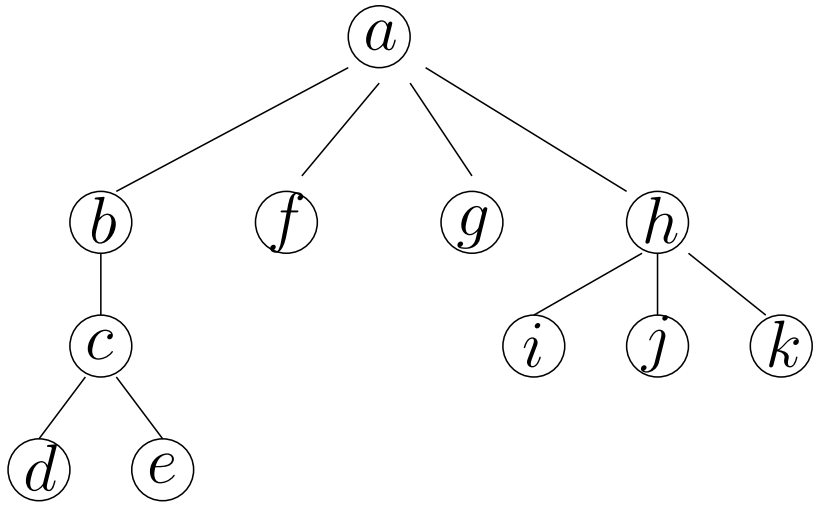
Rooted Tree

Consider a tree T and a specified node r in T.

The **rooted tree** of T at r, denoted by T_r, is the tree obtained by the breath-first search traversal of T starting from r; and r is the *root* of T_r.



a tree



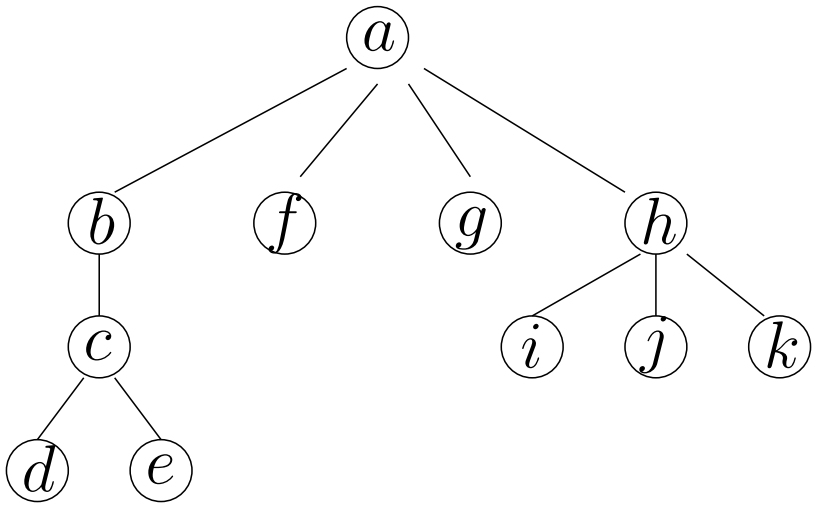
a rooted tree at node a

Child and Parent

Consider a node v *other than the root* in a rooted tree; let u be the adjacent node of v that is closer to the root. We say u is **the parent** of v, and v is **a child** of u.

Leaf

A node v in a rooted tree is a **leaf** if and only if v has no child.



a is the parent of h, while i, j and k are all leaf children of h.

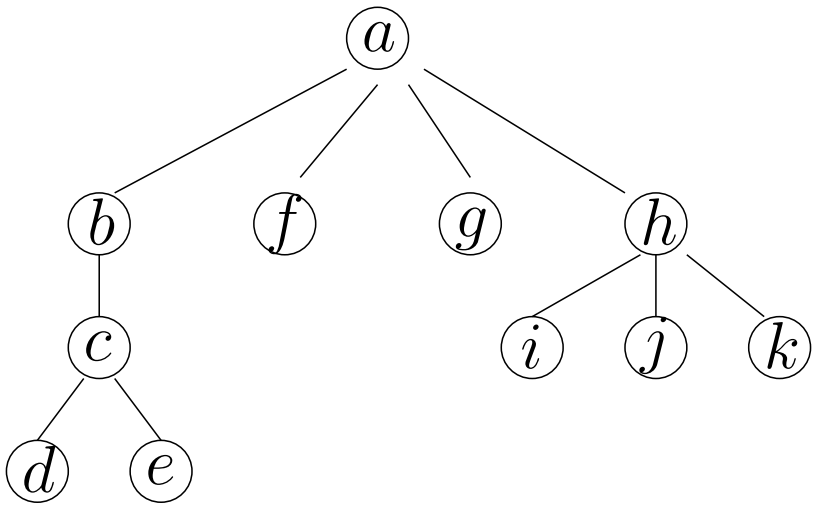
Sub-tree

Consider a node v *other than the root* r in a rooted tree T.

Removing the edge between v and its parent, the rooted tree T is divided into two (rooted) trees: one is rooted at r and the other rooted at v.

The one rooted at v is defined as the **sub-tree** of T rooted at v.

Moreover, we define: the sub-tree of T rooted at the root r is T itself.



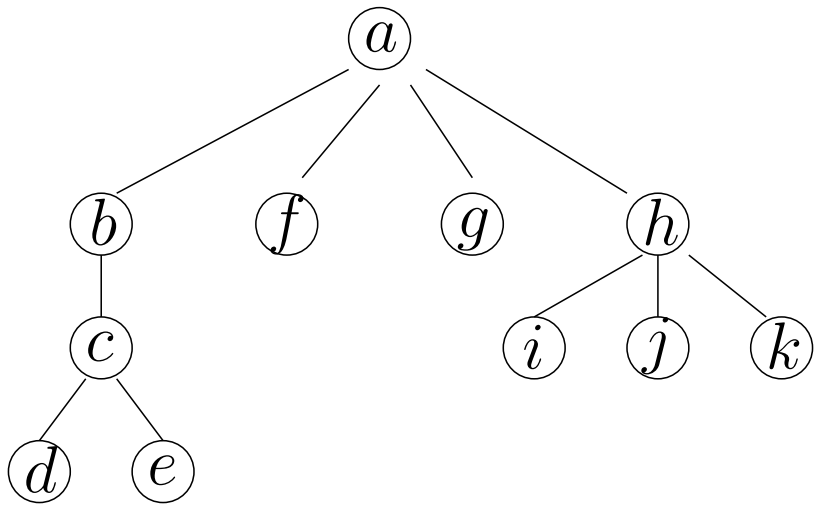
The tree consists of nodes $\{b, c, d, e\}$ is the sub-tree (of T_a) rooted at b.

Ancestor

For a node v in a rooted tree, all the nodes on the (simple) path from v to the root are **ancestors** of v. Except v itself, those nodes are v's **proper ancestors**.

Descendant

For a node v in a rooted tree, all the nodes in the sub-tree rooted at v are **descendants** of v. Except v itself, they are v's **proper descendants**.



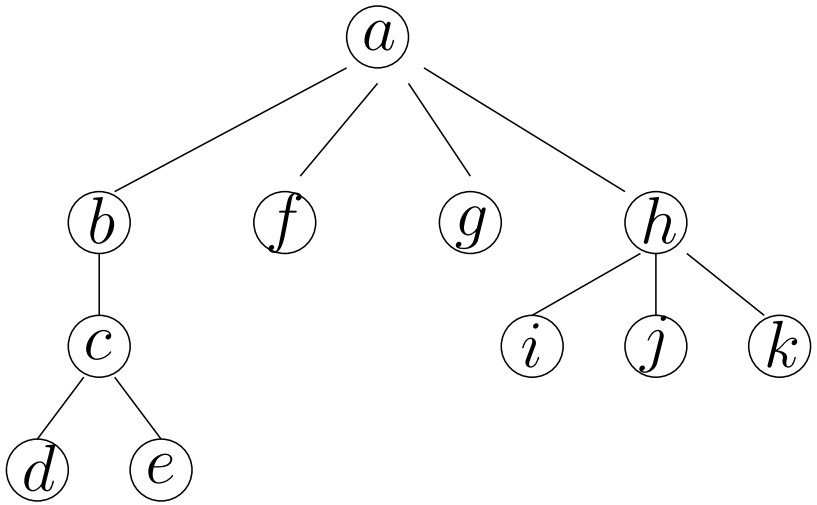
Nodes a, b and c are all ancestors of c, while a and b are c's proper ancestors.

Node Depth

The **depth** of a node v in a rooted tree is the number of *proper ancestors* of v. In particular, the depth of the root is 0.

Tree Height

The **height of a rooted tree** is defined as one plus the maximum depth of the nodes, namely, the number of nodes on the longest *root-to-leaf path* in the tree.



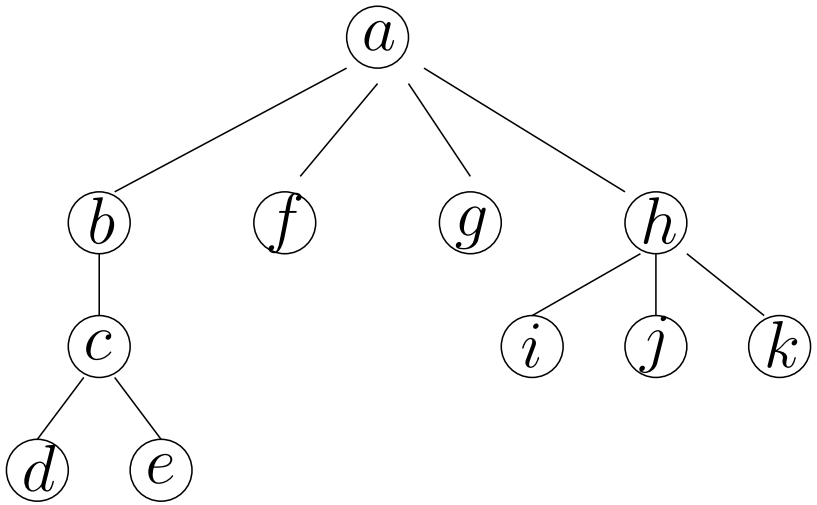
The maximum depth of the nodes is 3, while the height of the tree is 4.

Minimum Heap

A **minimum heap** (**min-heap** for short) on *a set of key values* is a rooted tree, where:

- each key must occur in at least one node in the tree; and
- for any node v in the tree, the key of v is *smaller than* the key of each of its children.

By definition, the root of any sub-tree of a min heap has the smallest key among all the keys of the nodes in the sub-tree.



A min-heap on the set of keys $\{a, b, \ldots, k\}$.

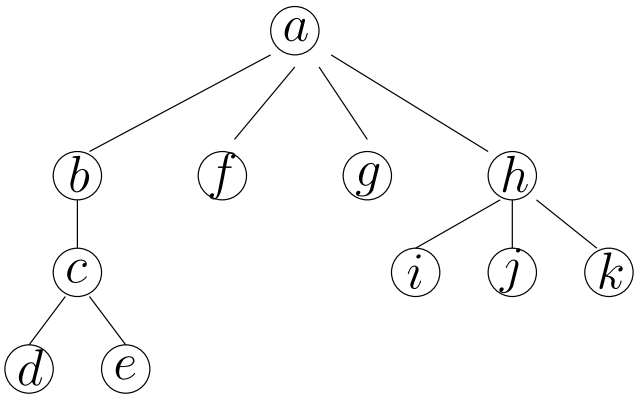
Preliminaries

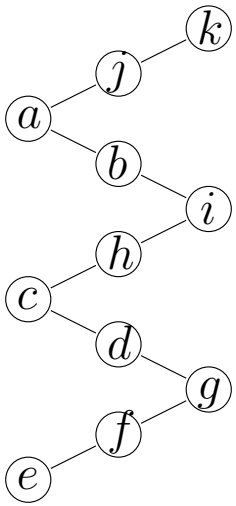
Binary Tree

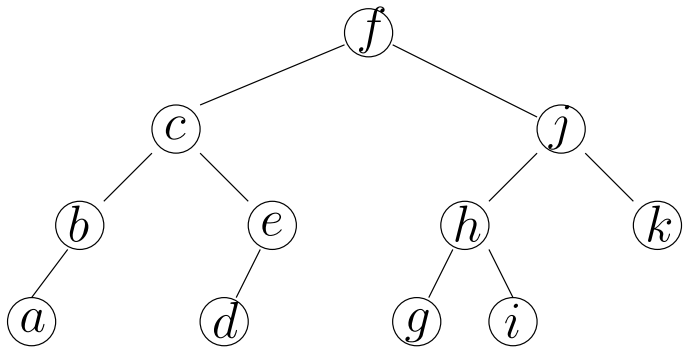
A **binary tree** is a rooted tree, where each node in the tree has *at most two* children.

Think:

Which of the following are binary trees?







Preliminaries

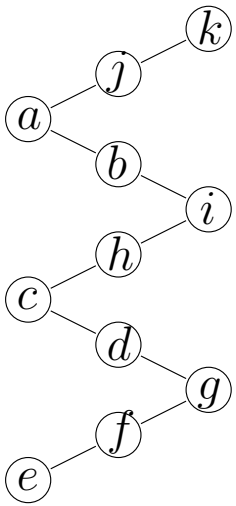
Binary Search Tree (BST)

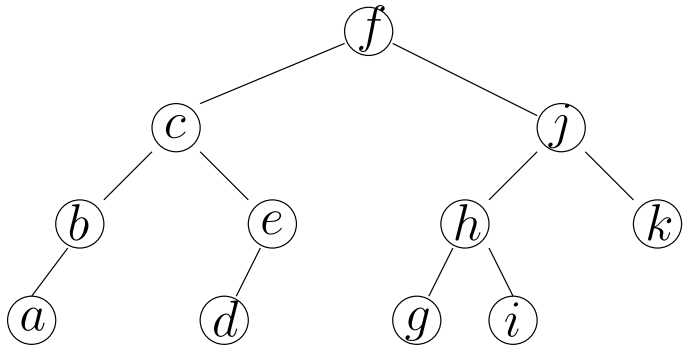
A **binary search tree** (BST) on a set S of n key values is a binary tree satisfying the following:

- Each key value occurs in at least one node of the tree.
- A node distinguishes its children: the left child and the right child;
- For any node v in the tree, the key of v is larger than *any* key of the nodes in its left sub-tree and smaller than *any* key of the nodes in its right sub-tree.

Think:

Which of the following are binary search trees? The letters are keys in alphabetical order.





Searching a key q_{key}, denoted by $\mathit{find}(q_{\text{key}})$, is one of the most important operations that a BST needs to support.

Specifically, the operation $\mathit{find}(q_{\text{key}})$ returns the node u in the BST whose key $\mathit{key}(u) = q_{\text{key}}$ if such a node u exists; and otherwise, returns **NULL**.

Think:

What are the worst-case searching length of $\mathit{find}(q_{\text{key}})$ of these BSTs? And what makes them different?

Balanced BST

The worst-case searching time of a BST actually **depends on** its height.

To reduce the worst-case searching time, we need to reduce the height of the tree, equivalent to **"balancing"** the tree.

When the BST also needs to support key insertions and deletions, this is particularly crucial.

With different **strategies** to maintain the balance, different balanced BSTs are defined, such as: **AVL Tree**, **Red-black Tree**, **(a, b)-Tree**, and etc. All these balanced BSTs achieve:

- $O(n)$ space;
- $O(\log n)$ worst-case time for each insertion, deletion or search.

In this lecture, we introduce an alternative randomized binary search tree, called **Treap**, which:

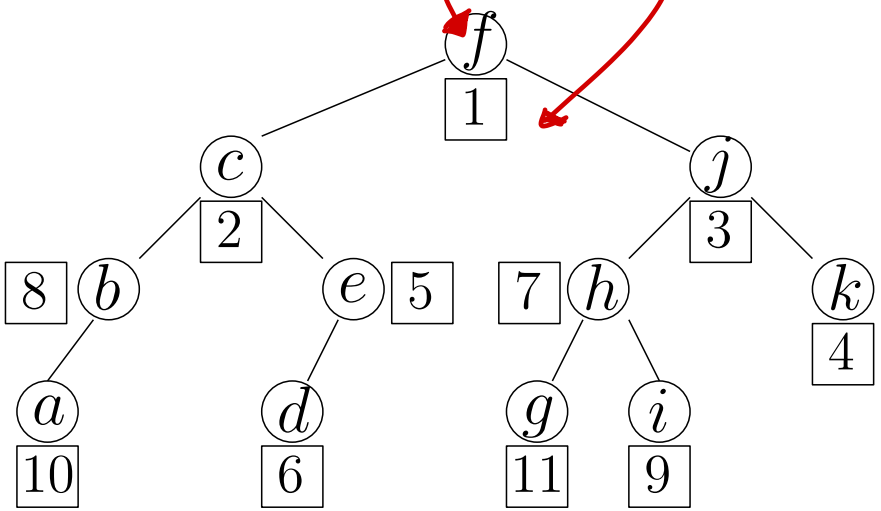
- consumes $O(n)$ space;
- supports each of several operations, e.g., insert, delete, find, join and split (which will be illustrated shortly), in $O(\log n)$ expected time.

The name "treap" comes from "tree" + "heap".

Treap

A **treap** is a binary tree, in which each node has a *search key* and a *priority*[1], satisfying the following simultaneously:

- it is a BST with respect to the *search keys*;
- it is a min-heap with respect to the *priorities*.

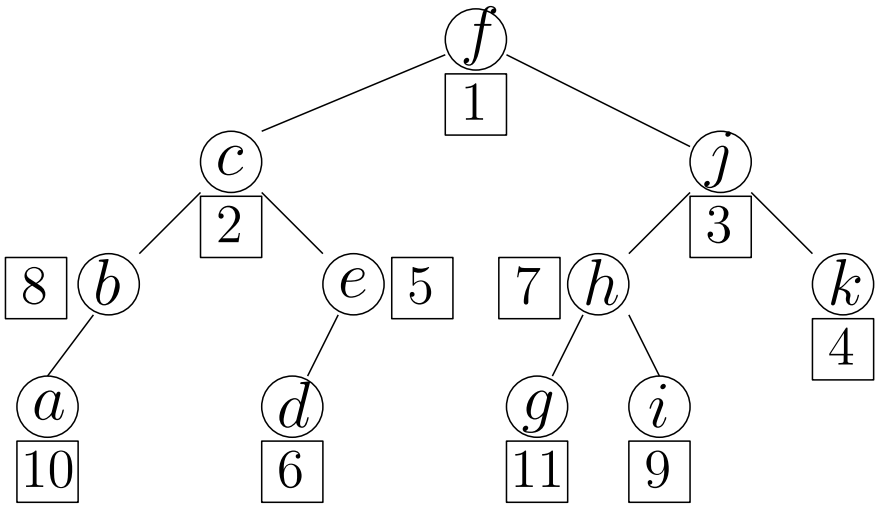


A treap, where the letters, in circles, are the search keys, and the numbers, in squares, are the priorities.

[1] For simplicity, we assume that all the search keys and priorities are distinct.

Treap

Observation 1. Every sub-tree of a treap is (also) a treap.



A treap where the letters in circles are the search keys, and the numbers in squares are the priorities.

Theorem 1. Given the search keys and the priorities, a treap is **uniquely defined**.

Proof.

We show this theorem by mathematical induction on the number, n, of search keys. First, when $n = 0$, the treap is empty and thus is unique. Suppose that the uniqueness of the treap holds for all $n \leq k$. Next, we consider a treap with $n = k + 1$ search keys.

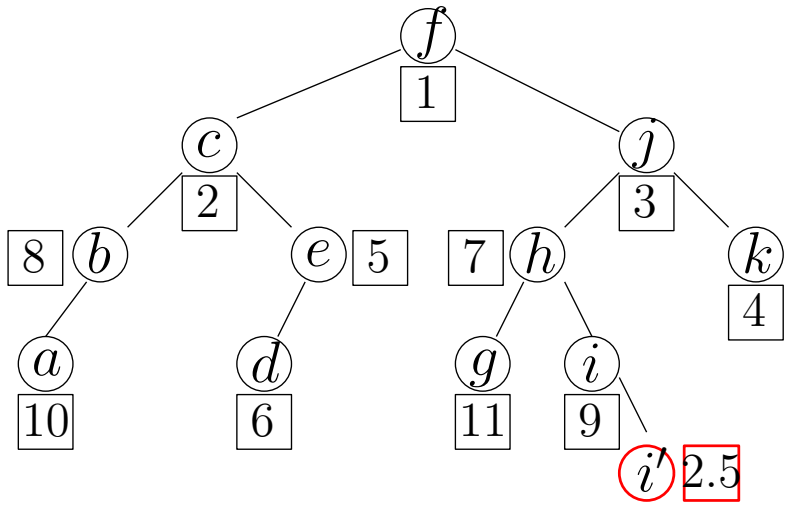
Since a treap is a min-heap on the priorities, the root node u of the treap must have the smallest priority. Moreover, as a treap is also a BST on the search keys, we know that the left sub-tree of u contains all the search keys $< key(u)$, and u's right sub-tree contains all the search keys $> key(u)$. As a result, the structure between u and its two sub-trees is unique.

By the inductive assumption, both the left and right sub-trees are treaps that are uniquely defined. The theorem follows. □

Heap Property Maintenance

Consider inserting a new search key, i', with priority 2.5, and with $i < i' < j$ into the treap.

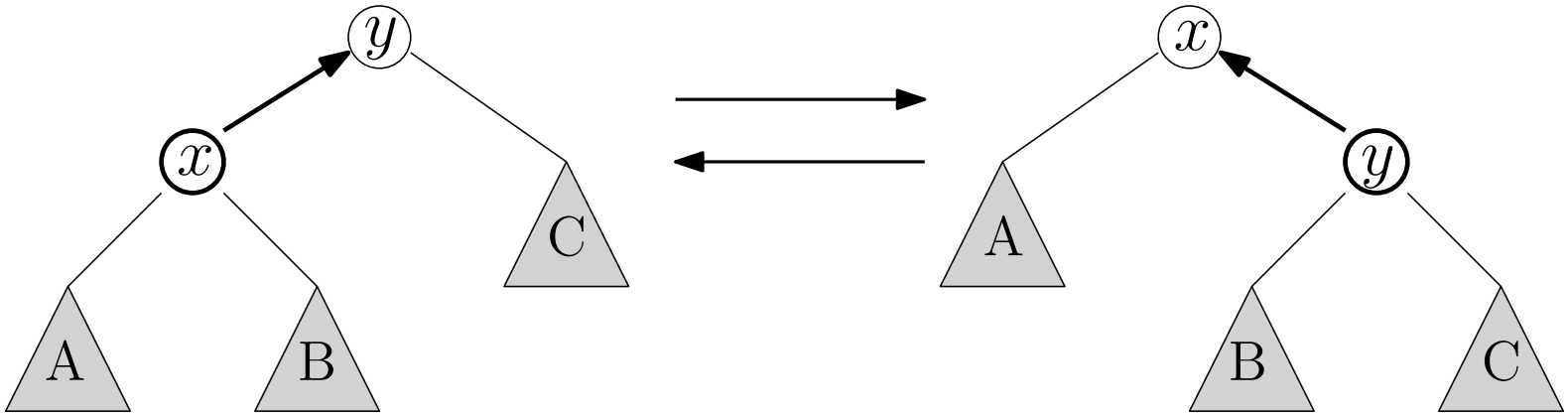
The BST aspect is easy: just insert i' as a leaf in the BST by the standard BST insertion algorithm, without changing the rest of the tree. However, after the insertion, the (min-)heap property is violated.



To resolve the violation, we rotate the node i' to a place where both the BST and heap properties are maintained simultaneously.

Rotations

The **left and right rotations** are two basic re-balancing operations adopted by many BSTs, such as AVL Trees and Red-black Trees.



a right rotation at x a left rotation at y

Each rotation can be performed in $O(1)$ time.

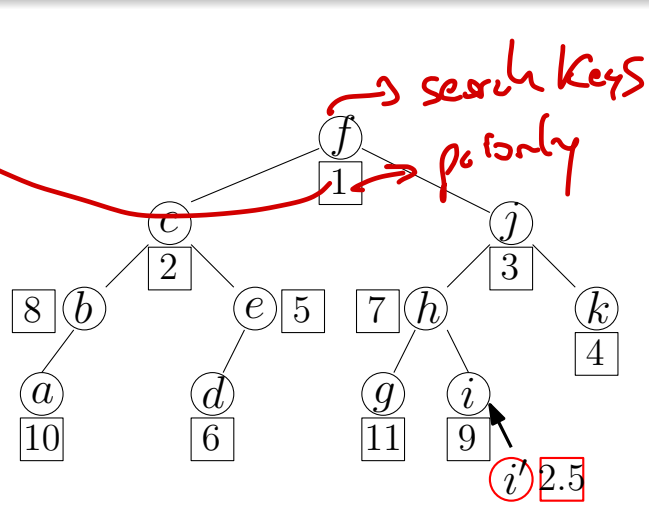
A nice feature of left and right rotations is that they do not violate the BST property: they can change the depth of a node and that of its parent.

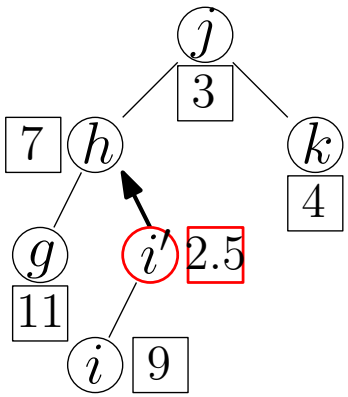
Therefore, we resolve min-heap property violations via rotations.

Treap

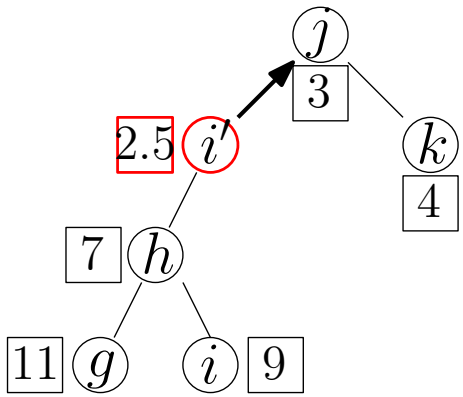
Heap Property Maintenance (Cont.)

We resolve the violation in the earlier example by rotations.

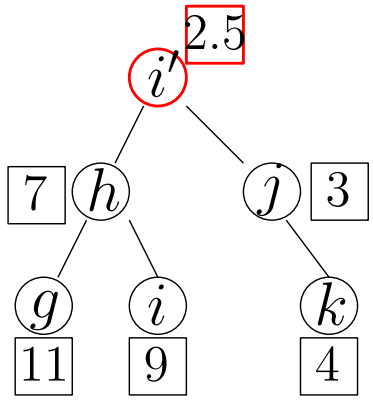




after the left rotation between i' and i



after the left rotation between i' and h



after the right rotation between i' and j

Reverse all the above rotations can rotate the node i' (back) to a leaf.

Treap Operations

Insertion

Insert a search key x with some priority:

- insert x to an appropriate leaf position in the BST;
- resolve the possibly heap property violations by rotations.

Think: What's the worst-case time complexity of this operation?

Cost: $O(height(treap))$

expected depth is $O(\log n)$

Treap Operations

Search

Search of key x: Invoke the operation $find(x)$ of the BST.

Cost: $O(height(treap))$

Deletion

Delete a search key x:

- find x in the BST (if x is not found, done);
- increase x's priority to ∞;
- resolve the heap property violations by rotations with the child node of x's which has smaller priority until x becomes a leaf;
- remove the leaf x.

Cost: $O(height(treap))$

Treap Operations

Join

Let $T_<$ and $T_>$ be two treaps, where all the search keys in $T_<$ are smaller than those in $T_>$.

Join $T_<$ and $T_>$ into a super treap:

- create a node u with priority ∞;
- make $T_<$ and $T_>$ the left and right sub-trees of u, respectively;
- resolve the heap property violation by rotating u to a leaf;
- remove the leaf u.

Cost: $O(height(treap))$

Treap Operations

Split

Let π be key value that does not belong to any node in a treap T.

Split a treap T into two treaps $T_<$ and $T_>$, such that all the keys in $T_<$ are smaller than π, and those in $T_>$ are larger than π:

- insert the search key π with priority $-\infty$; as a result, π will be the root u of the treap;
- take the left sub-tree of the root u as $T_<$, and the right sub-tree of the root u as $T_>$;
- remove the root u.

Cost: $O(height(treap))$

The cost of each operation on a treap T is proportional to $height(T)$. However, if T has n nodes, in the worst case, $O(height(T)) = O(n)$. (**Question:** Can you think of the worst case?)

Nonetheless, next we show that as long as the priority of each search key is uniformly and independently drawn from a continuous[2] range $[0, 1]$, the **expected cost** of each treap operation is bounded by $O(\log n)$.

To bound the expected cost of each treap operation, it suffices to bound the **expected depth** of *every* node.

[2] The continuity requirement is just to avoid the subtle case that two search keys have the same priorities.

Randomized Treap

Bounding the Expected Depth

Denote by u_k the node in the treap with the k^{th} smallest search key, for $k = 1, \ldots, n$. That is,

$$u_1 < u_2 < \ldots < u_{n-1} < u_n \,.$$

We use notation $i \uparrow k$ to denote the event that the node u_i is a proper ancestor of u_k.

Let $[i \uparrow k]$ be the indicator of whether event $i \uparrow k$ happens, i.e., $[i \uparrow k] = 1$ if the event happens, and 0 otherwise.

Randomized Treap

Bounding the Expected Depth

The definition of the tree node depth tells us that the depth of u_k is equal to the number of proper ancestors of u_k. Thus, we have:

$$depth(u_k) = \sum_{i=1}^{n} [i \uparrow k].$$

Taking the expectation of both sides, we have:

$$E[depth(u_k)] = \sum_{i=1}^{n} E[i \uparrow k] = \sum_{i=1}^{n} \Pr[i \uparrow k]. \quad (1)$$

Thus, it suffices to bound $\Pr[i \uparrow k]$ for all i and k.

Randomized Treap

Bounding $Pr[i \uparrow k]$

Define $U(i, k)$ as the subset of the treap nodes $\{u_i, u_{i+1}, \ldots, u_k\}$ such that:

- if $i < k$: $u_i < u_{i+1} < \ldots < u_k$, or
- if $i > k$: $u_k < u_{k+1} < \ldots < u_i$.

Lemma 1. For $i \neq k$, $[i \uparrow k] = 1$ if and only if u_i has the smallest priority among all the nodes in $U(i, k)$.

Bounding $Pr[i \uparrow k]$

Lemma 1. For $i \neq k$, $[i \uparrow k] = 1$ if and only if u_i has the smallest priority among all the nodes in $U(i, k)$.

Proof of the Only-If Direction.

- Since $[i \uparrow k] = 1$, u_i is a proper ancestor of u_k: in the tree rooted at u_i, u_k is in either the left sub-tree (if $i > k$) or the right sub-tree (if $i < k$).
- In either case, by the BST property, all the nodes in $U(i, k)$ are in the sub-tree rooted at u_i.
- Therefore, the min-heap property says u_i has the smallest priority among all nodes in $U(i, k)$.

□

Bounding $Pr[i \uparrow k]$ (Cont.)

Proof of the If Direction.

- We prove this direction via contradiction.
- Suppose that $[i \uparrow k] = 0$, that is, u_i is not a proper ancestor of u_k. There must exist a common ancestor, of u_i and u_k: the treap root is a common ancestor of them.
- Let u_j be the deepest common ancestor of u_i and u_k.
- Hence u_i and u_k must be in two different sub-trees, rooted at the two children of u_j; and the priority of u_j is smaller than those of both u_i and u_k.
- Furthermore, according to the BST property, u_j must be in $U(i, k)$.
- This contradicts the fact that u_i has the smallest priority in $U(i, k)$. Thus, $[i \uparrow k] = 1$ must hold.

□

Randomized Treap

Bounding $Pr[i \uparrow k]$ (Cont. 2)

Observation 2. As the priority of each node is uniformly and independently drawn from $[0, 1]$, each node in $U(i, k)$ is equally likely to have the smallest priority.

By Observation 2, we have:

$$Pr[i \uparrow k] = \begin{cases} \frac{1}{k-i+1} & \text{if } i < k \\ 0 & \text{if } i = k \\ \frac{1}{i-k+1} & \text{if } i > k \end{cases} \tag{2}$$

Bounding $E[depth(u_k)]$

Finally, substituting Equation (2) into Equation (1), we have:

$$\begin{aligned}
E[depth(u_k)] &= \sum_{i=1}^{n} \Pr[i \uparrow k] \\
&= \sum_{i=1}^{k-1} \frac{1}{k-i+1} + \sum_{i=k+1}^{n} \frac{1}{i-k+1} \\
&= \sum_{j=2}^{k} \frac{1}{j} + \sum_{j=2}^{n-k+1} \frac{1}{j} \\
&\leq \ln k + \ln(n-k+1) \\
&\leq 2 \ln n
\end{aligned}$$

Theorem 2. The expected cost of each treap operation of a randomized treap is bounded by $O(\log n)$.