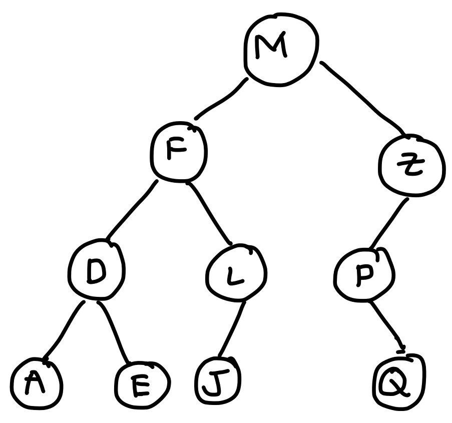
LECTURE 11 (October 2nd)

Randomized Binary Search Trees & Treaps

A binary search trees is a binary tree where each node is labeled with a key and key at any node is larger than the left subtree and smaller than the right subtree.

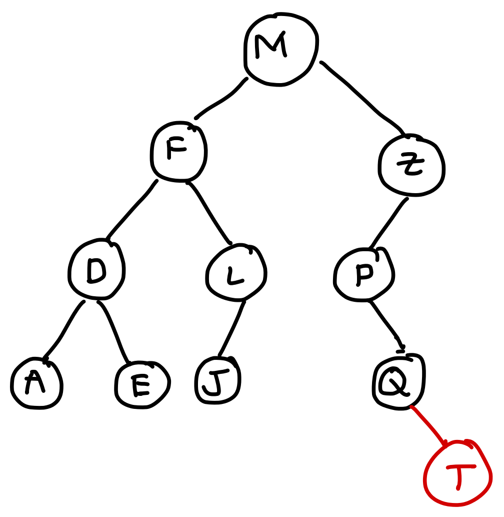
For example, if we use letters for keys, here is a binary search tree



__Find__ To find a key in a binary search tree, we just do binary search

__Insert__ To insert we just do a binary search, the search fails but whatever leaf we ended the search with, we insert the new node as a child of that leaf

For example, insert T in the binary tree above gives



Ideally, binary search trees should be balanced, and if our tree is balanced, the time to search takes $O(\log n)$ time, since the depth of every node is $O(\log n)$ where n = # nodes in the tree

The problem is that not every binary tree is balanced, for instance, the tree could have depth n where all nodes are to the right. In this case, search takes $O(n)$ time.

Moreover, we would like the tree to not be static, but instead we would like to insert / delete elements or do other operations.

Some examples of operations we would want

- Find (x) – Find a key x in the tree
- $\left.\begin{array}{l}\text{Pred}(x)\\ \text{Succ}(x)\end{array}\right\}$ – Find the predecessor or succesor if the search is unsuccessful

- Insert (x) or Delete (x) – Insert or delete
- Split (x) – Split a tree into two trees, one where all keys are $< x$ and other where all keys are $> x$.
- Join (x) – Reverse of the split operation

And ideally, what we would want is that all of these operations happen in $O(\log n)$ time

There are ways of doing this e.g. AVL or Red-Black trees, but these are quite complicated In fact, there was a bug in the C++ library that implemented red-black trees for close to 10 years. There are really subtle things that can happen.

Today, we are going to see a very simple way to do the above that uses randomness. In fact, once you see the rules, you can implement it yourself fairly easily.

Treaps – introduced by Aragon & Seidel in the 90s

The object we will look at is called a treap (= tree + heap) which is both a binary search tree and a priority heap.

Recall that heap is a data structure where every node has a priority and for a min-heap a node with a smaller priority is always an ancestor of a node with a higher priority.

For a treap, all the above operations will take $O(\log n)$ expected time (or even $O(\log n)$ time with high probability)

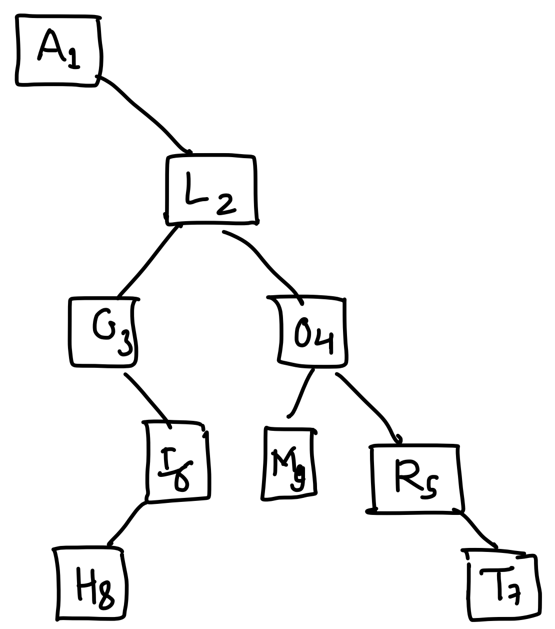
For a treap, each node has a key (which we will denote by letters A, B, C, D, ...) and a priority (which we will denote by natural numbers 1, 2, 3, ...)

If we only look at the keys, it looks like a binary search tree and if we only look at the priorities, it looks like a heap

As an example, consider $A_1 L_2 G_3 O_4 R_5 I_6 T_7 H_8 M_9$ (Priority 1 is smaller priority)

Assuming all the keys & priorities are distinct, there is exactly one tree corresponding to it.

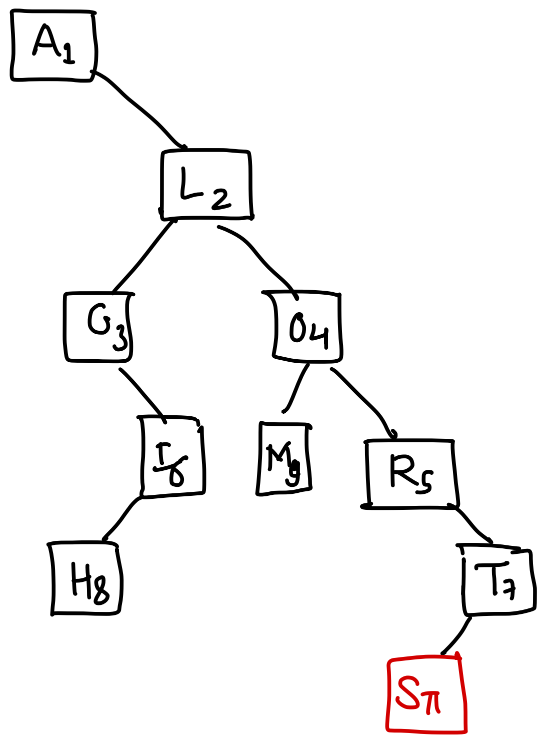
For the example above, the following is the treap



Item with the smallest priority is the root
To the left, are all letters less than A
To the right, are all letters greater than A
By induction hypothesis, each of the (key, priority) pairs define a unique tree

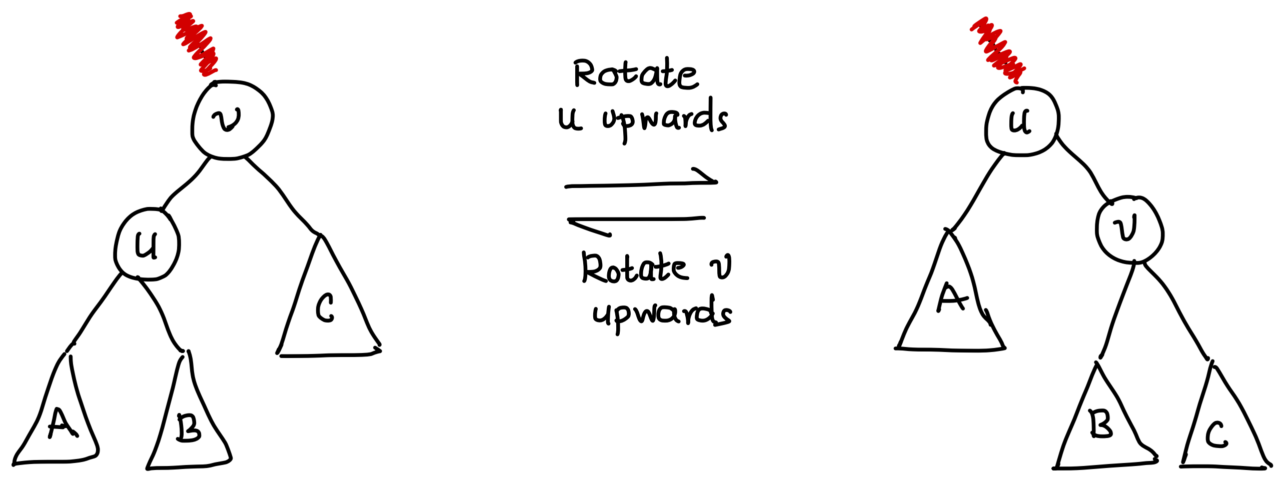
Let's see how to insert an element, e.g. S_π where $\pi = 3.14159 \cdots$ is the priority and S is the key

First look at the key and insert it in the search tree.



But now the priorities are not in the correct order!

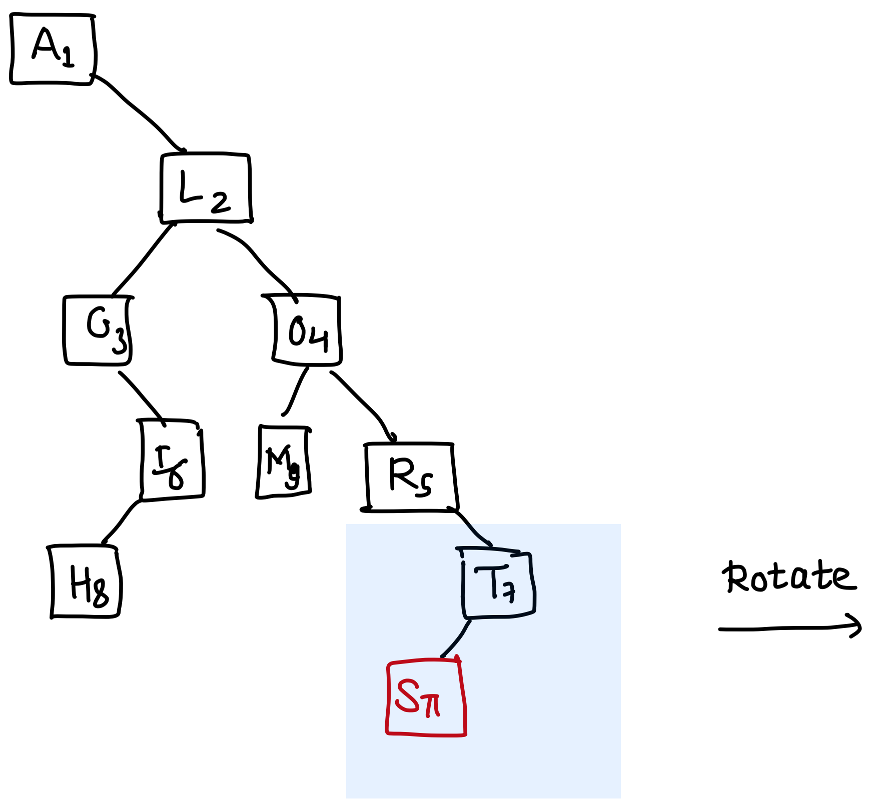
To fix the priorities, we use an operation called rotation (same operation is used in red-black or AVL trees)

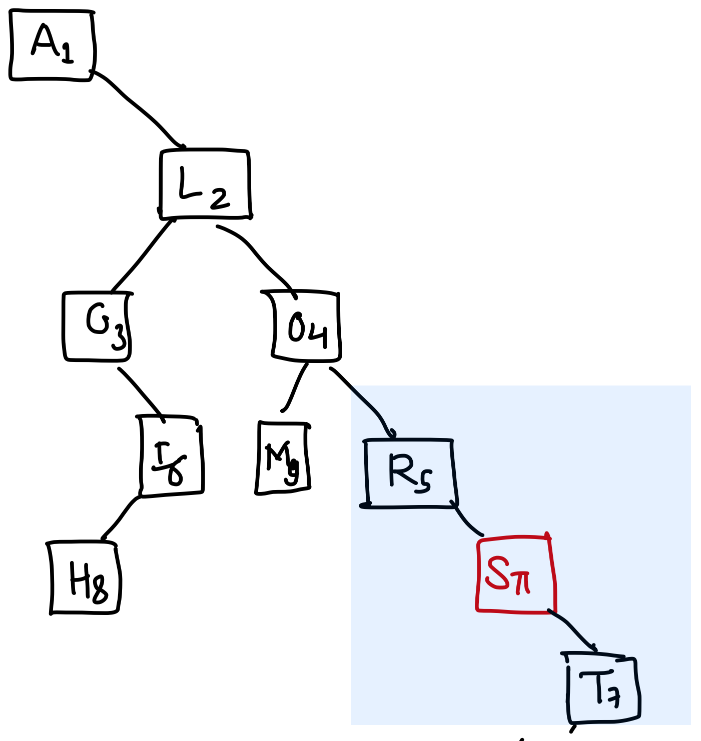


Rest of the tree remains unchanged

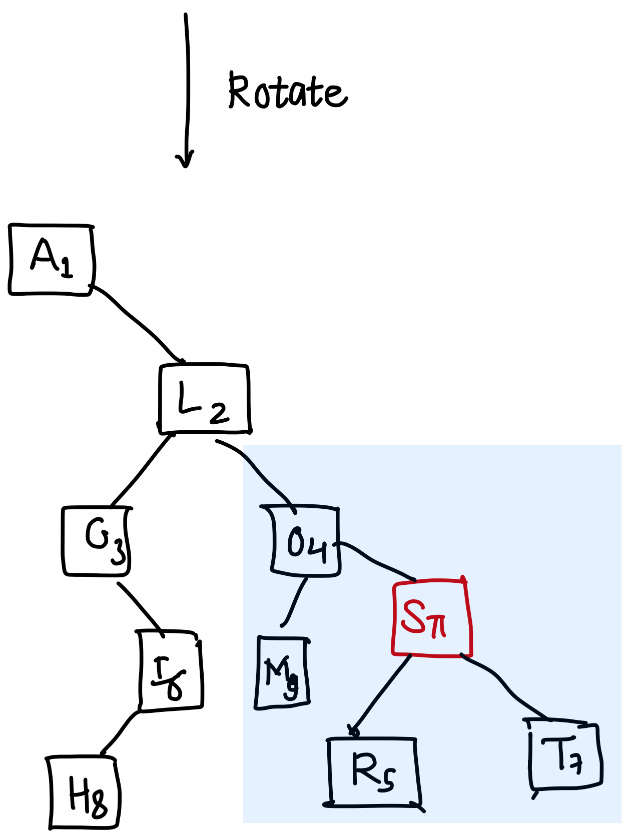
So, this can be done in O(1) time — just careful pointer manipulations

But one thing to note here is that if u was a child of v before, it is a parent of v afterwards. So, if priority (u) was smaller than priority (v) we have moved it upwards. This is exactly what we want to fix the priorities after insertion.

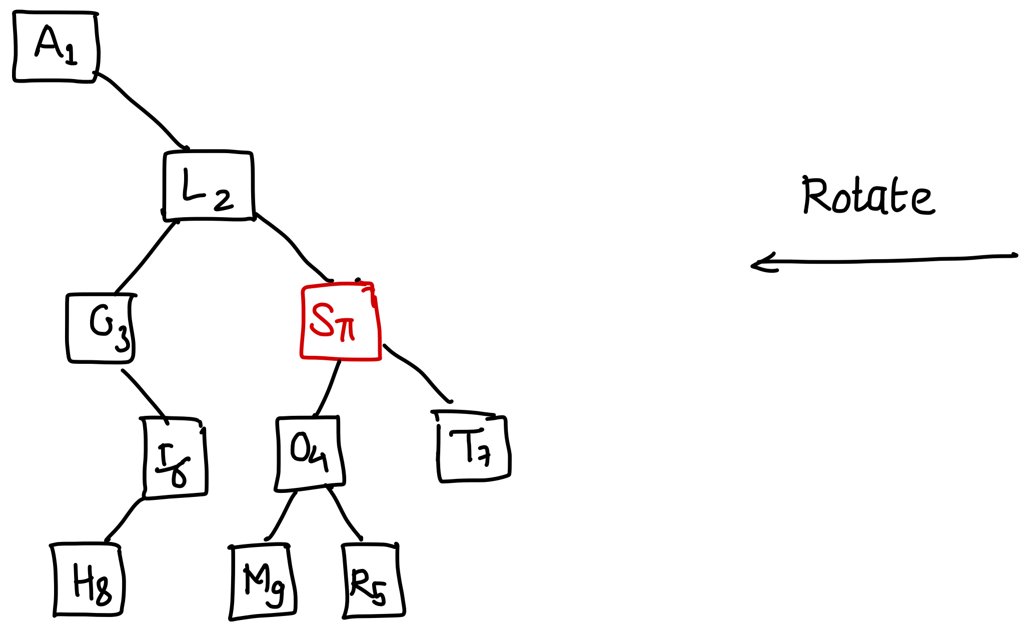




This is still not fixed, so we rotate again



Still not correct!



Now all of our priorities are correct!

So, to insert we pretend its a binary search tree, insert the node according to its key and then do rotation. Each rotation fixes the problem at that node but may move it closer to the root, so we keep doing it until all the priorities are correct. Other parts of the tree are not affected.

Insert (k, p) Create New node (k,p)
Insert only looking at keys
Bubble up new node using rotations only looking at priorities

How do we $\boxed{\text{delete}}$? Just do insertion in reverse order
Make the priority of that node ∞
Bubble down the node only looking at priorities
(Which node moves up is determined by the priority - smaller priority child moves up)
When it becomes a leaf, just remove it

Delete
Set priority to ∞
Rotate down (promoting child with smaller priority closer to root)
Remove the leaf

Run time of Insertion / Deletion is determined by depth of the initial/final inserted leaf

Run time of search is determined by the depth of the node the search ends on

So, For all of these operations, time $= O(\text{depth of some node})$

Recall, that the original motivation was to design a balanced binary search tree where we only have keys. So, the priorities can be chosen by the data structure itself since they are not part of the input.

How do we assign priorities? Just generate a uniformly random number in $[0,1]$ for each node independently. That will be its priority.

The random priorities then determine the structure of the binary search tree.

Main message If we choose the priorities randomly, then for every node v

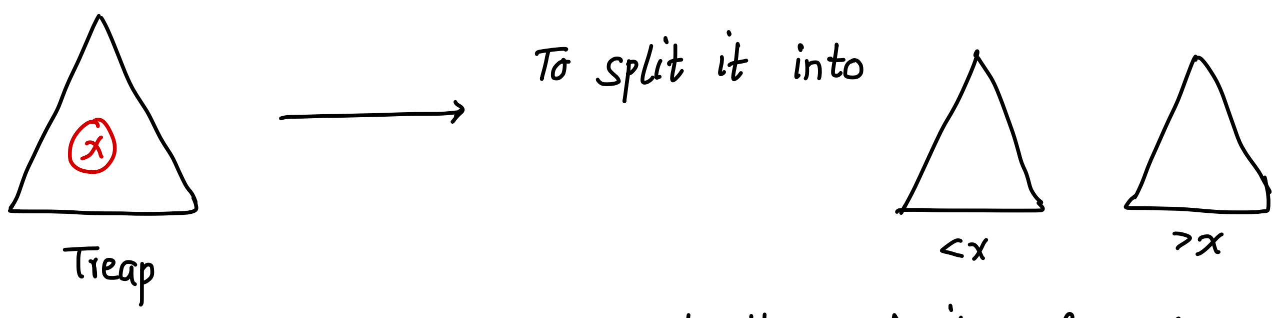
$$\mathbb{E}[\text{depth}(v)] = O(\log n)$$

In fact, in the next lecture we will see that $\mathbb{E}\left[\max_v \text{depth}(v)\right] = O(\log n)$ as well

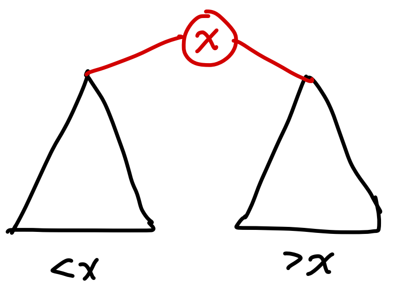
That is, the binary search tree is balanced in expectation (or with high probability even), and the running time of the above operations is $O(\log n)$ in expectation.

Split/Join Operations - How do we split/join?

Suppose we want to split at x.
If x is not in the treap, first insert it.



we set the priority of x to $-\infty$
Bubble it up until it becomes the root



And then delete the root

For joining, we just do this backwards in time

Time for these operations is also O(depth of some node)
so this is still $O(\log n)$ time assuming the statement about expected depth

Let's prove this statement now :

Main message If we choose the priorities randomly, then for every node v

$$\mathbb{E}[\,\text{depth}(v)] = O(\log n)$$

To simplify notation, we will assume that keys are integers $1, 2, 3, \ldots, n$
Also note that, depth (k) = # proper ancestors of k
↳ This means depth of node whose key is k

Key idea Express depth(k) as a sum of indicator random variables

$$\text{depth}(k) = \sum_{i=1}^{n} \mathbb{1}[\,i \uparrow k\,] \qquad \text{where } \mathbb{1}[i \uparrow k] = \begin{cases} 1 & \text{if } i \text{ is a proper ancestor of } k \\ 0 & \text{o/w} \end{cases}$$

Thus, $$\mathbb{E}[\text{depth}(k)] = \sum_{i=1}^{n} \mathbb{E}\left[\mathbb{1}[i \uparrow k]\right]$$

$$= \sum_{i=1}^{n} \mathbf{P}[\underbrace{i \uparrow k}_{\text{= Event that } i \text{ is a proper ancestor of } k}] \qquad \text{since } \mathbb{E}\left[\begin{matrix}\text{indicator}\\ \text{variable}\end{matrix}\right] = \mathbf{P}\left[\begin{matrix}\text{indicator}\\ \text{variable} = 1\end{matrix}\right]$$

What is the probability that i is a proper ancestor of k ?

If $i=k$, $\mathbb{P}[i \uparrow k] = 0$ since i can not be a proper ancestor

Lemma For all $i<k$, i is a proper ancestor of k iff priority (i) is the smallest priority among all nodes in $\{i, \dots k\}$ i.e.

$$i \uparrow k \iff \text{priority}(i) = \min\{\text{priority}(j) \mid j \in \{i, \dots k\}\}$$

Now, each node gets an independent random priority, e.g.

Priority	0·01	0·99	0·71	0·42		0·37
Key	1	2	3	4	-------	n

So, Assuming this lemma is true,

$$\mathbb{P}[1 \uparrow n] = \frac{1}{n}$$ since each node gets an independent random priority, so out of all n nodes, the probability that **1** has the smallest priority is $\frac{1}{n}$

$$\mathbb{P}[i \uparrow k] = \frac{1}{k-i+1}$$ (if $i<k$) since there are exactly $k-i+1$ priorities in that set out of which one has to be chosen to be the smallest one & they are all equally likely

$$\text{Thus, } \mathbb{E}[\text{depth}(k)] = \sum_{i=1}^{n} \mathbb{P}[i \uparrow k]$$

$$= \sum_{i=1}^{k-1} \mathbb{P}[i \uparrow k] + \sum_{i=k+1}^{n} \mathbb{P}[i \uparrow k]$$

$$= \sum_{i=1}^{k-1} \underbrace{\frac{1}{k-i+1}}_{\text{Above lemma}} + \sum_{i=k+1}^{n} \underbrace{\frac{1}{i-k+1}}_{\text{Symmetric lemma where we reverse the roles of } i \,\&\, k}$$

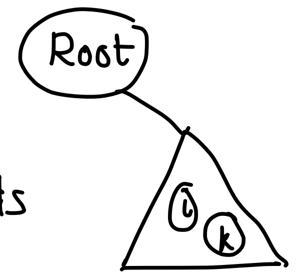
$$= \sum_{j=2}^{k} \frac{1}{j} + \sum_{l=2}^{n-k-1} \frac{1}{l}$$ by setting $j = k-i+1$ & $l = i-k+1$

$$= H_k - 1 + H_{n-k+1} - 1 = O(\log n)$$

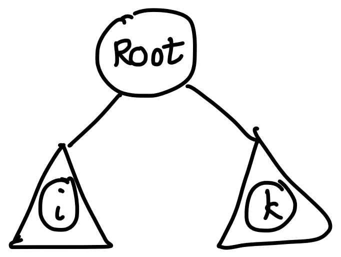
So, once we prove the lemma, we are done.

Proof of Lemma There are five cases to consider based on the root, keeping in mind that root has the smallest priority

- Root $< i$ $\longrightarrow$ If the root is smaller than i, then Both i & k are in the right subtree, so by induction hypothesis, the lemma holds



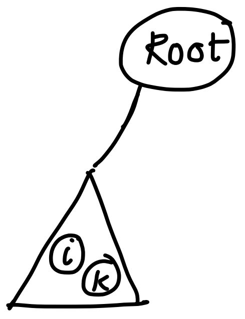
- Root $= i$ $\longrightarrow$ $i \uparrow k$ and priority (i) is minimum since its the root
- $i <$ Root $< k$ $\longrightarrow$ In this case, we know priority (i) is not minimum since it must be the root. We also know that



So, neither is a proper ancestor of the other

So, in this case, i is not a proper ancestor of k. i.e. $i \not\uparrow k$

- Root $= k$ $\longrightarrow$ $i \not\uparrow k$ here as well, since k is an ancestor of everything
- Root $> k$ $\longrightarrow$ Then both i & k are in the left subtree Lemma follows from induction hypothesis



■

So, we have now seen the definition of a treap, how to implement it and a complete analysis of expected depth.

Since, this is a lot simpler, why bother with AVL or red-black trees? Because for the latter the constant in $O(\cdot)$ is smaller, so if efficiency is very important then you should implement those o/w treaps are much easier to implement

One last thing to take away is that one can think of treaps in different ways. These all give equivalent & different ways of defining a treap.

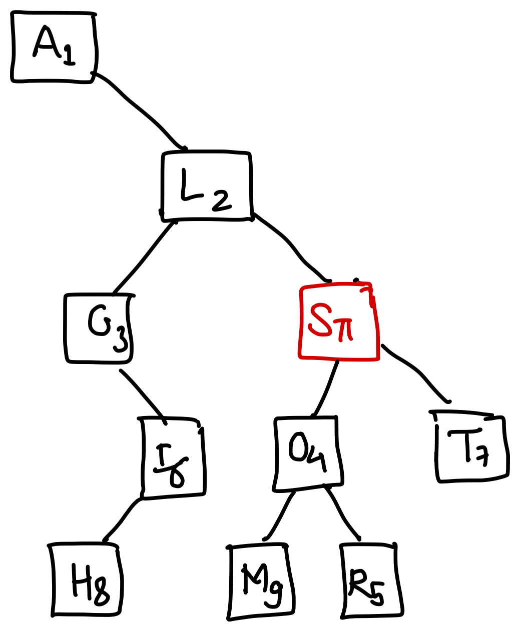
Definition 1 Treap is a binary search tree with random priorities

Definition 2 Treap is a binary search tree obtained by inserting keys in random order

Assigning random priorities & inserting it in order of priorities is the same as inserting keys in a random order

Definition 3 Treap is a recursion tree for randomized quicksort

Let's look at an example here : suppose we want to sort the word ALGORITHM in alphabetical order, we pick a random pivot say A (using random priorities / random order) Recursively sort everything smaller than A & everything greater than A.



The recursion pattern we get is exactly this binary tree where in the right subproblem, we first chose the pivot L & recursed.

So, what we have seen today is an analysis of randomized quicksort.

RANDOMQUICKSORT [A)

```
Randomly permute A
For each $x \in A$
    Insert $x$ in binary search tree
Do an inorder traversal of binary search tree
```

The comparisons we do in the above are the same as picking a random pivot & comparing to others

$$\mathbb{E}[\text{run time}] = \mathbb{E}\left[\text{time to insert all } n \text{ keys in a treap}\right] = O(n \cdot \log n)$$

So, when we are inserting into a treap, we are running quicksort and realize we are missing something & we go backward in time & fix it by doing as little work as needed. There are many applications of this in data structure where we record something in a recursion tree & go back in time and fix the recursion tree (e.g. git)