451: Splay Trees

G. Miller, K. Sutner
Carnegie Mellon University
2020/09/17

Total Recall: BST

In this lecture, by a binary tree we mean a rooted, ordered, binary tree T:

rooted: there is a special root node.

ordered: the children of a node are ordered left to right.

binary: there are at most 2 children.

For a binary search tree (BST) we have additionally a node labeling $\lambda : V \to A$ into some ordered set. For simplicity, assume all labels are distinct. As usual, we may occasionally (often) conflate vertices and labels.

Critical Condition:

An in-order traversal of the tree produces an ordered sequence.

Keep It Shallow

We can search in a BST in time depth of the tree (length of the longest branch).

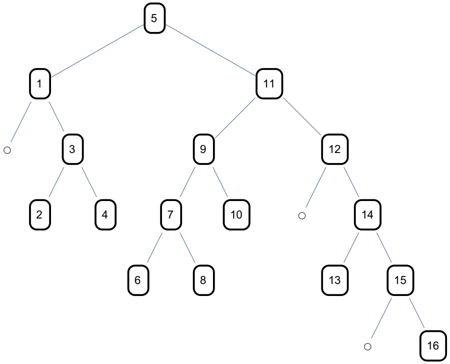
Hence, if the tree has size n and is reasonably balanced, then we can search in in $O(\log n)$ steps.

If, during a sequence of insertions, the tree becomes too unbalanced we need to work on rebalancing it. So the challenge is to keep the additional cost of rebalancing low.

If the tree is built at random it can be expected to the shallow:

Random Is Good

If the inserts are random, there is nothing to worry about.



$5, 11, 9, 12, 7, 8, 1, 3, 6, 14, 15, 13, 4, 2, 10, 16$

Digression: Expected Depth

Theorem

The average depth of a binary search tree generated by inserting a permutation of $[n]$ is $O(\log n)$.

Proof.

Define a random variable for the total path length in a binary tree:

$$X_n = \sum_{x \in T} \mathsf{depth}_T(x).$$

Here BST T was generated by inserting a random permutation of $[n]$. Write $e_n = \mathsf{E}[X_n]$ for the expected total path length in T, so the expected path length is e_n/n.

Let

$$p_k = \Pr[\text{ left subtree of root in } T \text{ has size } k\ ]$$

for $k = 0, \dots, n-1$. Then $p_k = \Pr[\lambda(\text{ root }) = k+1] = 1/n$.

Hence

$$\begin{aligned} e_n &= \sum_{i<n} p_i(n-1+e_i+e_{n-i-1}) \\ &= n-1+1/n\sum_{i<n}(e_i+e_{n-i-1}) \\ &= n-1+2/n\sum_{i<n}e_i \end{aligned}$$

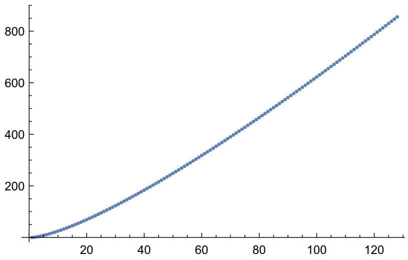
and so

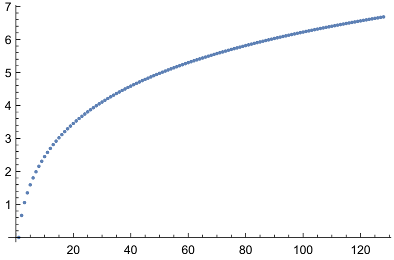
$$n\,e_n = n(n-1)+2\sum_{i<n}e_i$$

Subtracting this from the same equation for $n+1$ we obtain

$$e_{n+1} = \frac{2n}{n+1}+\frac{n+2}{n+1}e_n$$

Picture





Looks good: total path length is about $n \log n$ and expected search length looks like a logarithm.

Claim: The last recurrence has an upper bound $e_n \le 2n \ln n$.

Proof. This can be shown by induction on n using an approximation for $\ln(n+1)$. To this end, recall that the nth harmonic number can be approximated as follows:

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + O(n^{-3}).$$

where γ is Euler's constant, $\gamma \approx .5772$. It follows that

$$\ln(n+1) \approx \ln n + \frac{2n+1}{2n(n+1)}$$

where the error is $O(n^{-3})$. This is enough to show that the bound works as advertised.

$\square$

A Surprise

An expert in solving recurrences could actually produce a closed form solution, albeit in terms of the digamma function:

$$2\big((n+1)\gamma - 2n + (n+1)\psi(n+1)\big)$$

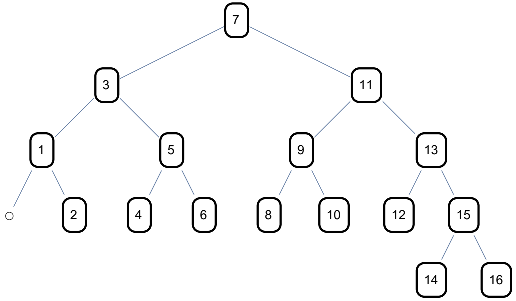
A few values:

$$0, 1, \frac{8}{3}, \frac{29}{6}, \frac{37}{5}, \frac{103}{10}, \frac{472}{35}, \frac{2369}{140}, \frac{2593}{126}, \frac{30791}{1260}$$

See polygamma for more information.

Forcing Balance

By contrast, some trees like AVL tree simply force the tree to be balanced. Of course, the extra work does not come for free.



Exercise

Review your old notes on BSTs and balanced BSTs.

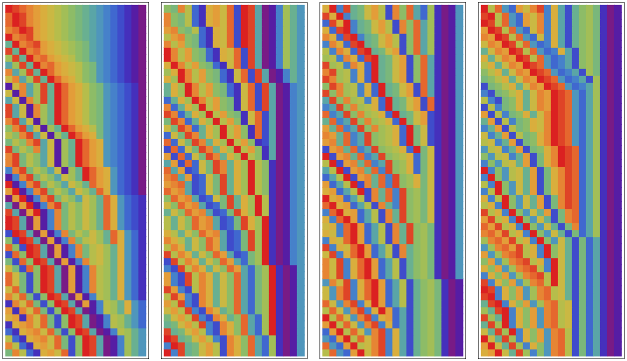
Here is a vague idea that may help to speed up container data structures: if an element x is accessed, move it to a place where it can be found more quickly, next time there is a request.

If we assume that the probability for requests for x is high, this should reduce overall costs.

First, a simple example: our data structure is a plain linked list of length n. Assuming random access, average cost is $n/2$.

Here is an improvement: move-to-front lists (MTF): after an access to x, move x to the front of the list.

Example



A list with 20 distinct elements, 200 access operations. Access probabilities ranging from 0.01 to 0.1.

Comparison

What is the proper reference performance in a static data structure?

Assume we are given the sequence of access requests in advance. Then we can compute access frequencies and sort the list accordingly. Call this optimal static list K and let L a MTF list with the same elements.

The expected cost for a search in K is $\sum_i i\, p_i$.

Example 1: For uniform probabilities this is $(n+1)/2$.

Example 2: But for $p_i = 2^{-i}$ we get expected cost around 1.085.

Simplify

To simplify notation, we will assume that $K = (1, 2, \dots, n)$. This is fine, we can simply rename the elements in the lists as well as in the access sequence (which clearly does not affect frequencies).

So $L = (\ell_1, \ell_2, \dots, \ell_n)$ is some permutation of $[n]$ that changes during the execution of the algorithm.

Exercise

Show that a list sorted by access frequencies is optimal as far as static lists are concerned.

Exercise

Verify the two examples on the last slide.

Intuitively, we need a potential function that somehow measures the distance between L and K. Alas, something simple-minded like Hamming distance won't work.

For all $x \in [n]$ define

$$\Phi_x = \#(\ z > x \mid\ z \text{ before } x \text{ in } L\ )$$

$$\Phi(L) = \sum_{z \in [n]} \Phi_z$$

This should look familiar: for Φ_x we are counting inversions in L with right endpoint x, and $\Phi(L)$ is just the total number of inversions.

For example, $\Phi(K) = 0$ but $\Phi(K^{\mathsf{op}}) = \sum_{i<n} i = n(n-1)/2$.

Now suppose $x \in [n]$ is in position k in L.

Claim: Accessing $x = \ell_k$ causes $\Delta\Phi = k - 1 - 2\,\Phi_x$.

To see this, note that the $k-1$ elements to the left of x can be grouped into s small and b big. So $k - 1 = s + b$. But $b = \Phi_x$, so $\Delta\Phi = s - b = k - 1 - 2\Phi_x$ and we have $-k < \Delta\Phi < k$.

For example

$10\,4\,5\,2\,9\,1\,7\,3\,8\,6 \to 8\,10\,4\,5\,2\,9\,1\,7\,3\,6 \qquad +4$

$2\,5\,8\,9\,1\,3\,4\,6\,7\,10 \to 1\,2\,5\,8\,9\,3\,4\,6\,7\,10 \qquad -4$

So?

Now consider an element $x = \ell_k$ in L. In the static list K, x is in position x and thus requires x steps to access.

$$\begin{aligned} \text{cost}_\Phi &= \text{cost} + \Delta\Phi \\ &= 2(k - \Phi_x) - 1 < 2x \end{aligned}$$

Now consider a long sequence of m operations from L_0 to L_m, say, $m = \Omega(n^{2+\varepsilon})$. Since the total difference in potential is bounded by $n(n-1)/2$, we have an amortized cost twice the cost in the optimal static list.

Exercise

What would happen if we simply transposed x with its left neighbor instead of moving it all the way to the front?

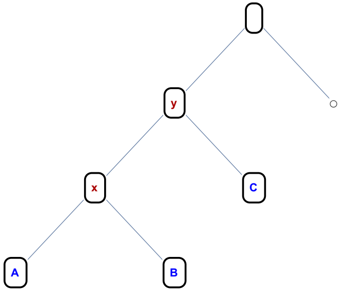
A splay tree is a BST, where every search for a node x is followed by a sequence of rotations that moves x to the root: we splay x. As a consequence, the tree remains reasonably balanced, though not in as rigid a manner as with other trees.

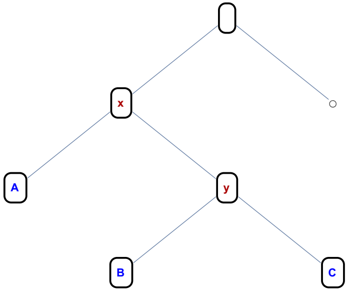
Alas, if this rotate-to-the-top operation is done blindly, the amortized cost could still be linear in the size of the tree.

Exercise

Show how ill-chosen rotations could fail.

Rotations





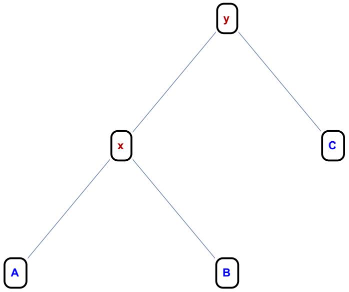
$\Longrightarrow$ right-rotate (about $x\ y$)

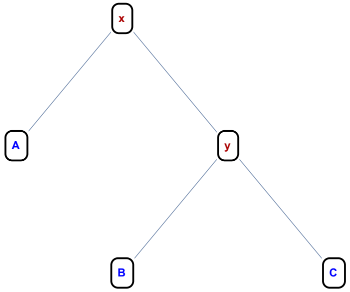
$\Longleftarrow$ left-rotate (about $y\ x$)

Note that rotations preserve the BST property.

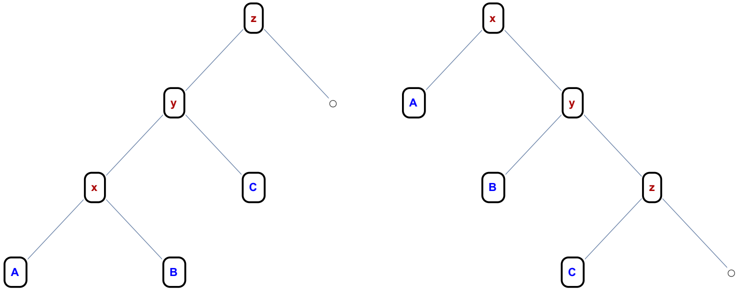
Suppose some vanilla search has found x. If x already is the root there is nothing to do.

If x is the left child of the root y, rotate about $x\,y$. This is called a zig.



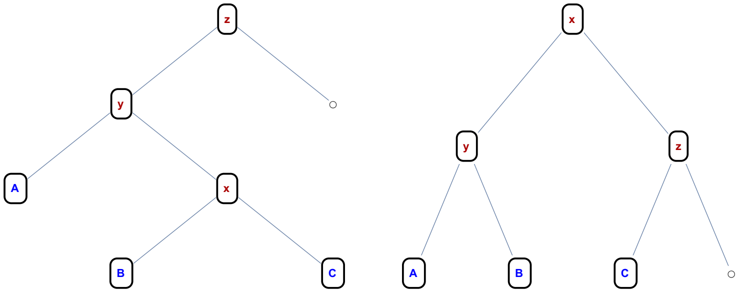


ZIG-ZIG



So there are two rotations, first about $y\ z$, then about $x\ y$.

Zig-Zag



Again two rotations, first about $x\,y$, then about $x\,z$.

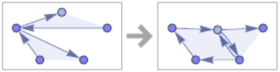
Of course, there are symmetric versions: zag, zag-zag, and zag-zig.

Note that these rules really form a graph rewrite system: given a suitable graph G (i.e., a binary tree), we can match the left hand side against some subgraph H and then replace it by the right hand side H', producing globally a new graph G'.

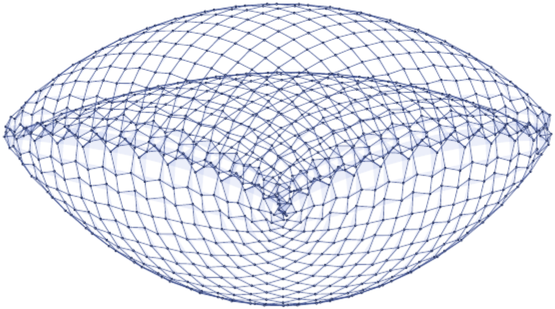
This is analogous to string rewriting: for example, context-free grammars such as

$$S \to \varepsilon \mid S(S)$$

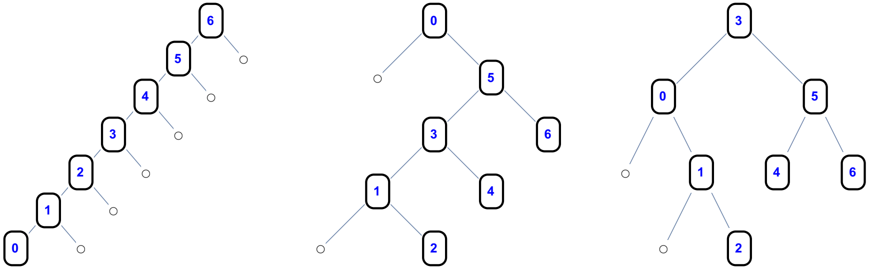
are string r/w systems. Alas, the technical details are significantly more complicated in the graph case, so we will rely on intuition rather than formal definitions.



gives after 1000 steps:

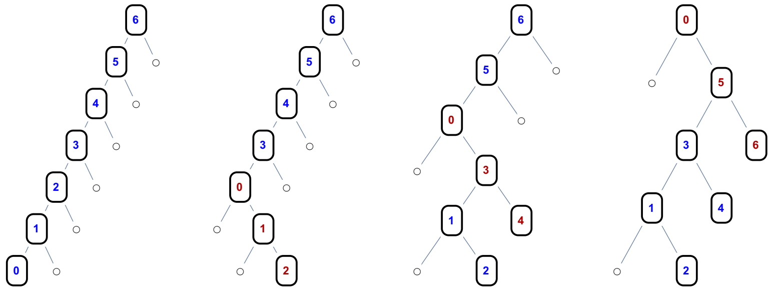


Two Splays



First we splay 0, then 3.

Splay 0 Steps



Exercise

Do the same step by step transformation for the following splay on 3.

Burning Question

Why should the splay rules produce amortized running time $O(\log n)$? It is entirely unclear that we could not wind up with lots of deep trees.

We will use a potential function to show that this actually cannot happen. Unsurprisingly, the right function is far from obvious here. Remember the Ansatz method?

We need to fix a bit of terminology: our BST will be T and we write T_x for the subtree with root x. We attach a weight $w(x)$ to each node and then use weights to define the potential.

Weights and Potential

$$W(x) = \sum_{z \in T_x} w(z)$$ size of x

$$\Phi_x = \lfloor \log W(x) \rfloor$$ rank of x

$$\Phi(T) = \sum_{z \in T} \Phi_z$$ potential of T

For us, $w(x) = 1$ so that $W(x) = |T_x|$ and the rank of x is essentially the logarithm thereof of the size of the tree.

However, on occasion it is better to use different weights, whence the general definition.

The potential of T is always the sum of all ranks.

But Why?

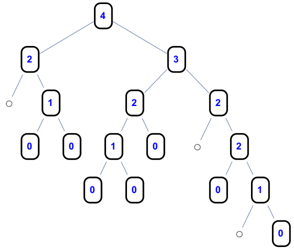
For example, for a complete binary tree T of depth d on $n = 2^d - 1$ nodes we have

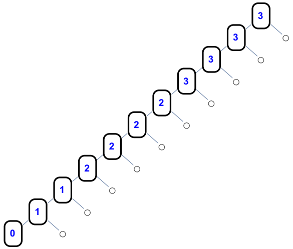
$$\Phi(T) = \sum_i i\, 2^{d-i} = 2^{d+1} - d - 1$$

For a degenerate path “tree” on $n = 2^k - 1$ nodes we have

$$\Phi(T) = \sum_i^{k-1} i\, 2^i = (k-2)2^k + 2$$

In general, the intent is that a balanced tree will have potential $O(n)$, but a very unbalanced tree will have potential $O(n \log n)$.





Potential 18 with 16 nodes on the left, potential 22 with 11 nodes on the right.

Exercise

Figure out what the potential of a degenerate one-branch tree is in general. Try some other simple shapes.

Access Lemma

Suppose we have a splay tree T of size n with root r.

Lemma (Access Lemma)

When splaying node x, the amortized cost cost_Φ *is bounded by* $3(\Phi_r - \Phi_x) + 1$.

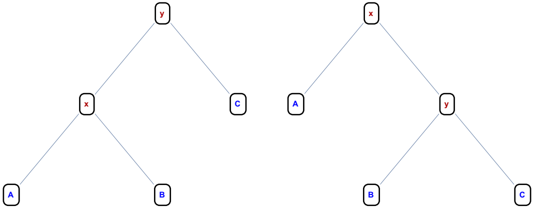
Proof.

So tree T is transformed into tree T' and $\text{cost}_\Phi = \text{cost} + \Delta\Phi$.

We need to consider the sequence of rotations involved with splaying x to the root.

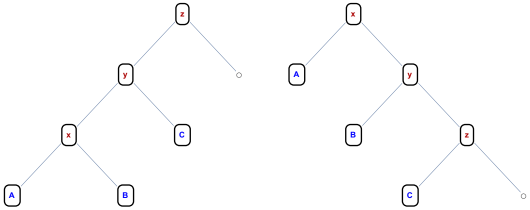
Alas, there are 3 (actually 6) possible cases: we have to determine $\Delta\Phi$ for zig, zig-zig and zig-zag.

Zig

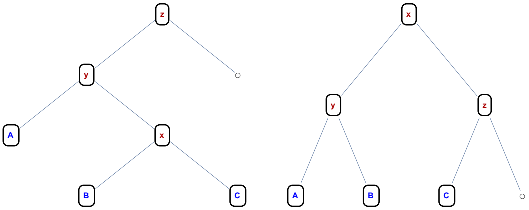


$$\begin{aligned}
\text{cost}_\Phi &= 1 + \Delta\Phi \\
&= 1 + \Phi'_x + \Phi'_r - \Phi_x - \Phi_r && \text{locality} \\
&\le 1 + \Phi'_x - \Phi_x && \Phi_r \ge \Phi'_r \\
&\le 1 + 3(\Phi'_x - \Phi_x) && \Phi_x \le \Phi'_x
\end{aligned}$$

Zig-Zig



$$
\begin{aligned}
\text{cost}_\Phi &= 2 + \Delta\Phi \\
&= 2 + \Phi'_x + \Phi'_y + \Phi'_z - \Phi_x - \Phi_y - \Phi_z && \text{locality} \\
&= 2 + \Phi'_y + \Phi'_z - \Phi_x - \Phi_y && \Phi_z = \Phi'_x \\
&\le 2 + \Phi'_x + \Phi'_z - 2\Phi_x && \Phi'_x \ge \Phi'_y, \Phi_y \ge \Phi_x, \\
&\le (2\Phi'_x - \Phi_x - \Phi'_z) + \Phi'_x + \Phi'_z - 2\Phi_x && \text{see claim 1} \\
&= 3(\Phi'_x - \Phi_x)
\end{aligned}
$$



$$
\begin{aligned}
\mathsf{cost}_\Phi &= 2 + \Delta\Phi \\
&= 2 + \Phi'_x + \Phi'_y + \Phi'_z - \Phi_x - \Phi_y - \Phi_z && \text{locality} \\
&\le 2 + \Phi'_y + \Phi'_z - 2\Phi_x && \Phi'_x = \Phi_z, \Phi_y \ge \Phi_x, \\
&\le (2\Phi'_x - \Phi'_y - \Phi'_z) + \Phi'_y + \Phi'_z - 2\Phi_x && \text{see claim 2} \\
&= 2(\Phi'_x - \Phi_x) \\
&\le 3(\Phi'_x - \Phi_x)
\end{aligned}
$$

Done

There may be many zig-zigs and zig-zags, but there is at most one zig operation during the whole sequence of rations while splaying x.

So, from the case analysis, we can pick up at most one term 1, the rest is a telescoping sum of $\Delta\Phi_x$ terms.

$$\begin{aligned} \textsf{cost}_\Phi(\textsf{splay } x) &\le 3(\Phi_r - \Phi_x) + 1 \\ &= O(\log |T|/|T_x|) \\ &= O(\log n) \end{aligned}$$

□

Auxiliary Claims

Claim 1: $2 \leq 2\Phi'_x - \Phi_x - \Phi'_z$

Claim 2: $2 \leq 2\Phi'_x - \Phi'_y - \Phi'_x$

Exercise

Verify these claims.

Corollary (Balance Theorem)

A sequence of m splay operations is $O((n+m)\log n)$.

Proof.

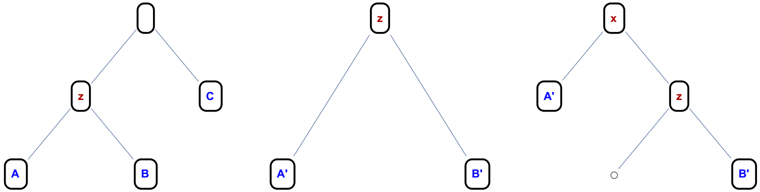
This is really a corollary to the Access Lemma:

$$\begin{aligned}\sum \mathsf{cost} &= \sum \mathsf{cost}_\Phi + \Phi(T_0) - \Phi(T_m) \\ &= O(m\log n + n\log n)\end{aligned}$$

□

Suppose we wish to insert x into T. Conduct a vanilla BST search to find a node z with left/right subtrees A and B such that, say, $A < x < z$.

Splay z to the top and then construct the tree shown below.



One can easily see that we get back a BST, but could this possibly wreck our amortized analysis?

By the Access Theorem we have amortized cost $O(\log n)$ for the vanilla search and the following splay.

The following tree surgery is obviously $O(1)$, but we need to be careful: the ranks change.

The size of T'_z can only be smaller than the size of T_z, so the rank of z can only decrease.

The new node x has size $n + 1$ and rank $\lfloor \log(n + 1) \rfloor$, which is certainly $O(\log n)$.

Delete

To delete node x, first splay it to the root, delete it, and join the two subtrees.

Join

Suppose we have two splay trees T_1 and T_2 and we wish to combine them where all nodes in T_1 are to the left of all nodes in T_2. Find the right-most element in T_1 and splay it to the root; then adjoin T_2 as right subtree.

Exercise

Convince yourself that these operations are amortized $O(\log n)$.

Comments

The version presented here is bottom-up splaying. There is an analogous top-down version where the rotations start at the root.

More material can be found at Splay Trees, including C code and further references. The code is surprisingly simple, given the major difficulties in the performance analysis.

Playing with Weights

We can prove other results by using different weights. Suppose item a_i is accessed $m_i \geq 1$ times, so $m = \sum m_i$.

Theorem

The total time for m operations is $O(m + \sum m_i \log(m/m_i))$.

Proof.

Change the weight to $w(x) = m_i/m$ so that $W(T) = 1$ and $\Phi_{\text{root}} = 0$.

Biggest potential change at x: x moves from root to leaf.

$\Delta\Phi_x = -\log m_i/m = \log m/m_i$

So the biggest total potential change in the whole tree is

$\Delta\Phi = \sum \log m/m_i$

Proof Continued

The cost for accessing x is

$$\begin{aligned}3(\Phi_r - \Phi_x) + 1 &= -3\Phi_x + 1 = -3\log W(T_x) + 1\\ &= 3\log\left(\frac{m}{\sum_{z\in T_x} m_z}\right) + 1\\ &\le 3\log\frac{m}{m_x} + 1\end{aligned}$$

Hence, the total cost for all operations is

$$\begin{aligned}\sum \mathsf{cost}_\Phi + \Delta\Phi &\le \sum m_i\left(3\log\frac{m}{m_i} + 1\right) + \sum \log m/m_i\\ &= O\left(\sum m_x \log\frac{m}{m_x} + m\right)\end{aligned}$$

$\square$