CS 240 — Lecture 10

Bartosz Antczak *Instructor: Eric Schost* *February 2, 2017*

10.1 AVL Tree Operations

- **Search:** just like BST, cost $\Theta(h)$, where h = height
- **Insert:** runtime to insert a node is in $\Theta(h)$. After that, `fix()` will be called once, which has a runtime of $\Theta(1)$. Total runtime = $\Theta(h)$.
- **Search:** First search for node to delete, then swap with the successor (or predecessor), then move up the tree and apply `fix()`. Total cost is $\Theta(h)$.

10.1.1 Height of an AVL Tree

Let's define $N(h)$ to be the *least* number of nodes in a height h AVL tree. We claim that the height of an AVL tree with n nodes is $\Theta(\log n)$. To show that, we'll prove:

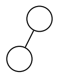
- $h \in O(\log n)$(*)
- $h \in \Omega(\log n)$ (**)

Proof of (*)

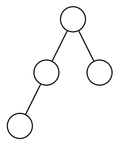
We call $N(h)$ the *least* number of nodes in an AVL tree of height h. Observe:

- $N(0) = 1$

- $N(1) = 2$



- $N(2) = 4$



In general, $N(h) = 1 + N(h-1) + N(h-2)$. Observe that this sequence looks like the Fibonacci sequence: $N(h) = F_{h+3} - 1 = \frac{\varphi^{h+3}}{\sqrt{5}} - 1$. Doing a bit of algebra:

$$N(h) \in O(\varphi^{h+3})$$
$$\log(N(h)) \in \log(\Theta(\varphi^{h+3}) = \Theta(h)$$

From this, we deduce that $h \in O(\log(N(h))$, and since $N(h) \leq n$, $h \in O(\log n)$.

Proof of (**)

We saw before that in a binary tree of height h, the number of nodes satisfies

$$n \leq 2^{h+1} - 1 \leq 2^{h+1}$$

Thus, $\log(n) \leq h + 1 \iff \log(n) - 1 \leq h$, ergo $h \in \Omega(\log n)$.

10.2 Dictionaries Part II

Recall that a *dictionary* is a collection of key-value pairs (KVPs). There are various ways to implement a dictionary:

- Unordered array or linked list
- Ordered array
- Balanced search trees

10.2.1 Optimal Static Ordering

Let's try to implement this dictionary through *optimal static ordering.* It's referred to as 'static' because this list is constructed before runtime (i.e., before a program is executed), and it's optimal because it reduces the runtime of searching for an element in the ordering.
The ordering involves placing the elements which are most likely to be accessed in the front of the list, this means that these elements require the least amount of time to find and they're the ones which will be searched for most often.

Expected Runtime

The expected runtime of this ordering is defined by the number of comparisons required to find any element in the ordering. The runtime depends on the respective probabilities of access for each element.

- If we have n entries with a **uniform distribution** (i.e., each element has an equally likely chance of being chosen), the expected number of comparisons, E_n, is

$$E_n = \frac{1}{n} + 2\left(\frac{1}{n}\right) + \cdots + n\left(\frac{1}{n}\right) = \frac{n+1}{2}$$

- If we have n entries with an **exponential distribution** (i.e., the preceding element is twice as likely to be accessed then the current element), the expected number of comparisons, E_n, is

$$E_n = \frac{1}{2} + 2\left(\frac{1}{4}\right) + 3\left(\frac{1}{8}\right) + \cdots + (n-1)\left(\frac{1}{2^{n-1}}\right) + n\left(\frac{1}{2^{n-1}}\right)$$

  Here, $E_n \in \Theta(1)$.

If the list was arranged in reverse as

$$\left[\frac{1}{2^{n-1}}, \frac{1}{2^{n-1}}, \cdots, \frac{1}{4}, \frac{1}{2}\right]$$

(where the element that is *least* likely to be chosen is first), then the number of comparisons, E'_n, is

$$E'_n = \left(\frac{1}{2^{n-1}}\right) + 2\left(\frac{1}{2^{n-1}}\right) + \cdots + (n-1)\left(\frac{1}{4}\right) + n\left(\frac{1}{2}\right)$$

This runtime is $E'_n \in \Theta(n)$.

10.2.2 Dynamic Ordering

What if we do not know the access probabilities ahead of time? Then we use **dynamic ordering**. This ordering involves two particular methods:

- **Move-To-Front (MTF):** upon a successful search, move the accessed item to the front of the list
- **Transpose:** upon a successful search, swap the accessed item with the item immediately preceding it

This ordering mutates the list based on past search results. As the name suggests it's a dynamic approach to the optimal static ordering.

10.2.3 Skip Lists

A **skip list** is a hierarchy of ordered linked lists. For a set of S items in a series of lists $S_0, S_1, \cdots, S_h$ such that:

- Each list S_i contains the special keys $-\infty$ and $+\infty$
- List S_0 contains the keys of S in non-decreasing order
- Every list contains all of the elements of every list preceding it (i.e., $S_h \subseteq S_{h-1} \subseteq \cdots S_0$)
- S_h contains only the two special keys

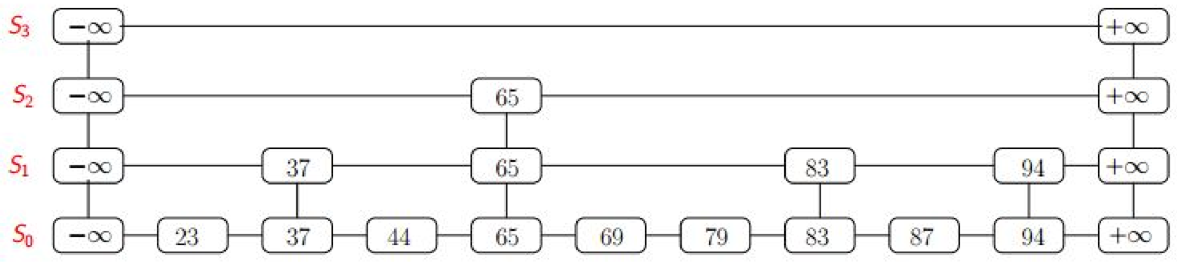


Figure 10.1: An example of a skip-list. Courtesy of CS 240 lecture slides.