CSC263 Notes

Data Structures and Analysis

https://github.com/ICPRplshelp/

Last updated January 29, 2023

1 Data Types, Data Structures

ADTs

- Specification
  - Objects we're working with
  - The operations (WHAT but not how)

Data structures

- Implementation (how)
  - Data
  - Algorithms

Analysis (runtime or complexity)

- Worst case
- Best case
- Upper bounds
  - $\mathcal{O}$
- Lower bounds
  - Ω
- Tight bounds
  - Θ

If we have algorithm A and input x, the runtime $t_A(x) =$ number of constant time operations independent of x.

Ultimately, we want a measure of running time that is a function of the input size. We have lots of inputs for each input size. So, if we want to prove an upper bound when looking for the worst case running time:

- I need two functions to prove an upper bound
  - A simple algebraic expression
  - The running time
    * However, the pure runtime function's codomain isn't $\mathbb{R}^{\geq 0}$ – but rather, a list of running times. To turn it into a raw function that outputs $\mathbb{R}^{\geq 0}$, we can take the largest of the list I just described.

Worst case is just us narrowing down a bunch of possible runtimes to the worst one.

My upper bound will always be some value that is larger or equal to the worst case, and the lower bound must be below the worst case but not all the worst cases.

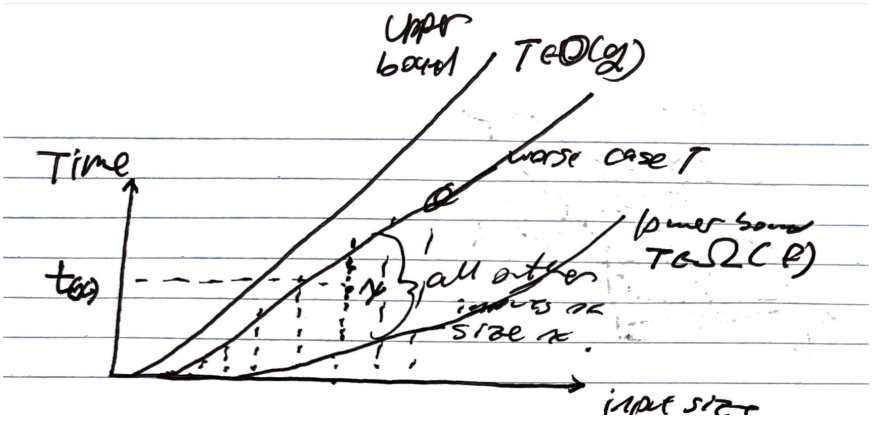

Figure 1: Upper and lower bounds of the worst case

1.1 Average-case Running Time

For a particular input size n, I have all these inputs x. We need to precisely define $S_n = \{\text{set of all inputs of size } n\}$. We need a probability distribution over our set of inputs.

- For each input $x \in S_n$, $t(x)$ is a random variable.
  - It's something that assigns a number to each element of our sample space. Now, S_n (a finite set) becomes a sample space.

Given a **discrete** probability distribution over S_n, $t(x) =$ the running time for input x. But what we'll get is that if I only know the input size, $t_n =$ **average-case running time for size n is:**

$$t_n := E[t(x)] \text{ over } S_n$$
$$= \sum_{x \in S_n} t(x) \Pr(x)$$

This summation is not as easy to figure out. There's a way we can get a particular value, but there is another way:

EX: The linear search algorithm

```
1 def LinearSearch(L: LinkedList, x: T) -> Node | None:
2     """Pre: L is a linked list, x is a value
3     Post: return the node that contains x
4     or none otherwise"""
5
6     z = L.head
7     while z != None and z.key != x:
8         z = z.next
9     return z
```

Standard linked list search.

Here's what we need to do:

Define our sample space (a family of sample space, one for each sample size):

Let n be arbitrary.

$$S_n = \{\text{every input of size } n\}$$

What should the sample space be? It shouldn't necessarily have infinitely many inputs with a given running time.

The number of different possible running times I have is finite for this algorithm.

INSIGHT. One representative input for each possible behavior. Behavior = running

time.

$$\begin{aligned} & S_3 \\ &= \{([1,\,2,\,3],\,1),\,([2,\,1,\,3],\,1),\,([2,\,3,\,1],\,1),\,([2,\,3,\,4],\,1)\} \end{aligned}$$

Alternatively, $S_n = \{([1,\,2,\,\ldots,\,n],\,1),\,([1,\,2,\,\ldots,\,n],\,2),\,\ldots\}$

Which is $\{([1,\,2,\,\ldots,\,n],\,x)\ : x \in [0,\,n] \wedge x \in \mathbb{N}\}$

The probability distribution becomes important. How do we decide how likely we want each input to be? How can we tell? That is a tricky question. What are we trying to do, and there's no obvious way to choose. In practice, in any kind of real-life scenario, if you want to judge how well an algorithm performs on average if you have some idea of what your real-life inputs are going to look like. If you have no information at all, where it is all abstract, then we'll just uniformly distribute.

$\Pr([1,\,2,\,\ldots,\,n],\,i) = \frac{1}{n+1}$ for $i = 0,\,1,\,\ldots,\,n$

Now we have this, we can calculate the expected value:

$$E\left[t(x)\right] = \sum_{(L,\,i) \in S_n} t(L,\,i) \cdot \Pr(L,\,i)$$

$$= \sum_{i=0}^{n} t\left([1,\,2,\,\ldots,\,n],\,i\right) \cdot \frac{1}{n+1}$$

When we're doing an average case, we cannot calculate an expected value with $\mathcal{O}$ expressions in there. We need a precise expression we can add up and average out. We need an **exact** expression

- Not in the sense that there's one right answer, but we need to fix a particular way of counting and count the same way for every input.

There is one trick: pick some representative operation that we know if we count that, the number of representative operations is $\Theta(\text{runtime})$.

In the example, the number of times `z.key == x` is run, which will be the **representative operation**

Ignore the constant time operations. The thing that matters is the loop. The loop does a constant amount of work each operation.

$$= t([1, 2, \ldots, n], 0) \cdot \frac{1}{n+1} + \sum_{i=1}^{n} i \cdot \frac{1}{n+1}$$

It's all algebra by this point. Do all of it, and you should end up with $\frac{n}{2} + \frac{n}{n+1} \in \Theta(n)$.

2 Priority Queues and Heaps

In a priority queue, we store a collection of elements. We're relying on **one** characteristic:

- Each element in the priority queue comes with its **own** priority attached to it (has something that makes each object sortable).
  - `x.priority` returns a comparable value
    * Integers are a good stand-in; however, we could use tuples or lists as a tiebreaker – the exception is that they are reversed for the context of this course. Hence, the last element takes the most precedence.
  - We don't care about its implementation. There is some built-in mechanism in the object that allows me to know its priority in constant time.

The **operations are the following:**

- `INSERT(Q, x):` add `x` to `Q`
  - Multiple elements can have the same priority.
  - If an object has multiple priorities, priorities assigned to the higher index should take precedence
- `MAX(Q):` return an element with the maximum priority.
  - If there are ties, we don't care which one is returned. Any of them could be returned. `Q` remains unchanged; this operation only queries our ADT.

- `EXTRACT_MAX(Q):` remove and return the element with the maximum priority.

The ordering for priority queues in this course will not be following the **first-in first-out** model. Drop the notion of a regular queue.
Of course, you could add a timestamp or insertion order as a tiebreaker.

2.1 How do we do it?

For the purposes of this course:

- An array will refer to an array-based list.
- A list will refer to a linked list.

The simplest data structures:

Unsorted array/list

- `INSERT`: $\mathcal{O}(1)$
- `MAX`: $\Omega(n)$
- `EXTRACT_MAX`: $\Omega(n)$
- We could do better.

Sorted array/list:

- `INSERT`: $\Omega(n)$
  - Yes, even for array-based lists. No loopholes.
- `MAX`: $\mathcal{O}(1)$
- `EXTRACT_MAX`: $\mathcal{O}(1)$

Here, $\mathcal{O}$ means "good news", and Ω means "too bad." It is simply emphasis. Everything is Θ, and all the time complexities here are **worst-case time.**

2.2 Max-Heaps

Intuition: partially sort

Structure: "almost complete" binary tree. **Everything full**, except maybe the last. On the last level, all leaves are **as far left as possible.** This **MUST** be always preserved throughout the lifetime of the max-heap.

In practice, when you have an almost-complete binary tree, the way that this is stored in memory, they generally mean a **list (or an array). They are listed in the order you would traverse then in a BREADTH FIRST SEARCH.**

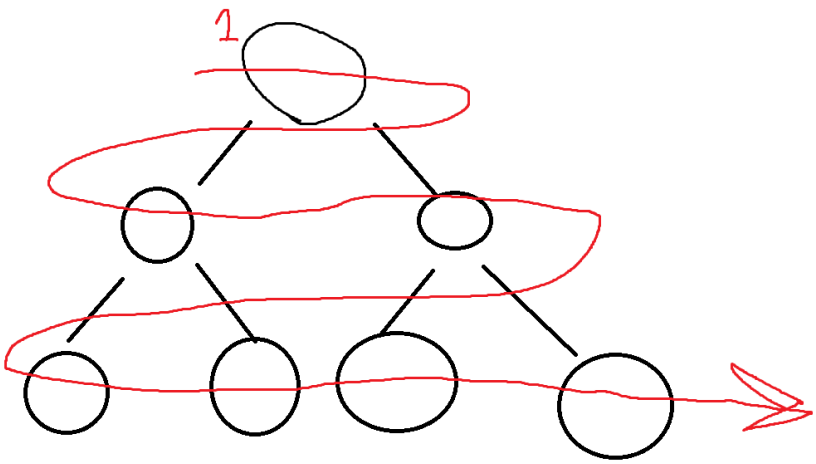

Figure 2: Indexing a heap

THE HEAP ELEMENTS START AT INDEX 1, AND WE SKIP OVER INDEX 0. Index 0 stores an empty item which we will not consider the root.

Navigation (USES BFS-LIKE ORDERING FOR INDEXING OF THE HEAP):

- For each node at index i in an array
- Parent$(i) = \lfloor \frac{i}{2} \rfloor$
  - This operation can be done extremely fast using bit shifts. That is, multiplication or floor division by a power of 2. In practice, this is implemented that way.
- LeftChild$(i) = 2i$
  - Previous item in size if it exists, otherwise we can't say anything
- RightChild$(i) = 2i + 1$
  - Next item in size if it exists, otherwise we can't say anything

Because we're working with an array (really), when we insert, we're going to add at the end. But it will mess up the ordering of the element. The same thing applies when we do extract max. We need to make sure that there are no gaps in the tree at any time.

2.3 Max-Heap Order

This is not a binary search tree. There is **no** left-to-right ordering. The only kind of ordering we have in a heap is the top-to-bottom ordering.

This property must hold for all max heaps.
EVERY NON-LEAF NODE STORES AN ELEMENT WITH PRIORITY $\geq$ THE PRIORITIES OF THE ELEMENTS IN THE NODE'S CHILDREN.
No required ordering between siblings. This means if I reflect the heap on the vertical axis, it shouldn't break this property.

View max-heaps as stairs: when you vertically go down, you should step downwards.

The highest priority of the heap is at the top. Finding the max is always easy: it is **always the top element, at index 1 (the first index).**

2.4 Heap Insertion

Insert at the end and try to push it up.

Insert any item into the heap.

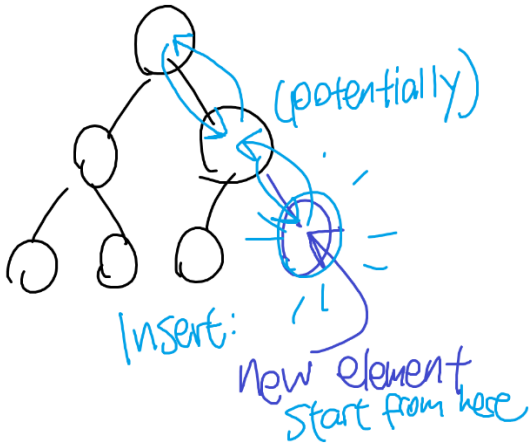

Figure 3: How insertion feels like

Do whatever we need to do to change things and preserve the entire tree structure. For `INSERT(n)`:

- Firstly, add `n` to the end. Our array would be: `[__, ..., n]`
  - Normally has extra space at the end due to array size inconsistencies
- `n` is a leaf. The only place that might be an issue is with the node and the parent.
- We swap the position of n with its parent, if $n >$ its parent.

- Then, check n with its parent (if it exists) and do the same thing as the previous step, again. Repeat that all the way up.

> Any point where `n.priority` $>$ `p.priority`, we swap. Keep doing this until that isn't the case.

Why does this not cause a problem with the neighbors: if s is a neighbor of X originally? Because $p \geq s$, and if $n > p$, then $n \geq s$.

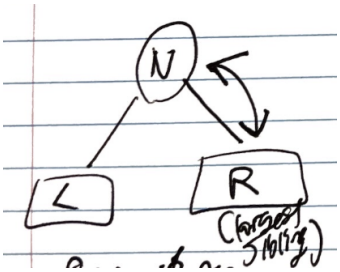

Figure 4: Largest sibling swap case

2.4.1 Heap Insertion Running Time

Constant work per level of the tree. We have a complete, balanced binary tree. The running time is $\Theta(\lg n)$, the height of the tree, for worst case.

2.5 Heap Extract Max

Move the last element to the top and heapify it (push it down).

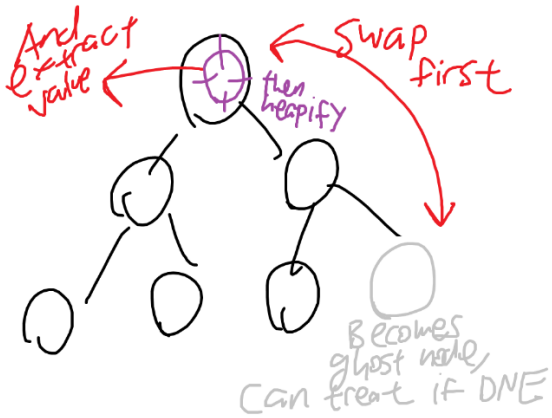

Figure 5: How extract max feels like

Remove and return the element in the heap with the highest priority. Procedure goes as follows:

- Decrease size of heap by 1: `H.size -= 1`
- Swap max value (index 1) with smallest value (index `H.size + 1`, the previous `H.size` before subtracting by 1)
- Perform Heapify on the root of the heap, now at index 1

2.6 Heapify

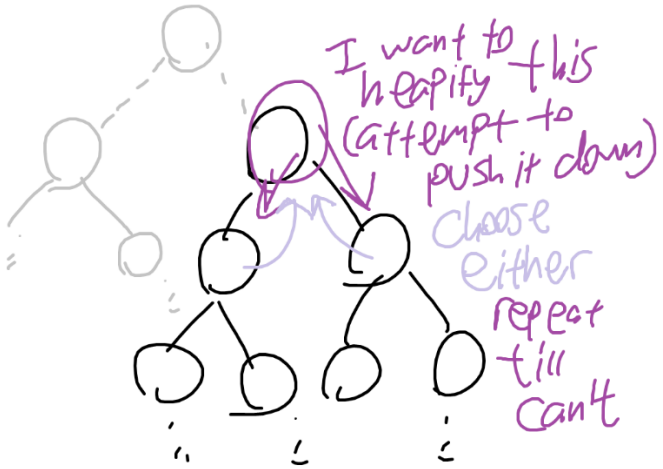

Figure 6: How it feels to heapify

Repeatedly swap element with its child of higher priority than the element until both children have smaller or equal priorities, or if I'm at a leaf.

Runtime is $\mathcal{O}(\lg(n))$. We do have to check all children (twice for a binary heap).

Heapify should only really be called if all subtrees are proper heaps.

2.7 Building a Heap

Turn the unsorted array into a heap, then run heapify on every index from right to left, starting at $\left\lfloor \frac{\text{len}(A)}{2} \right\rfloor$.

I have a whole collection of objects, and I want to put them into a heap.

- Start from collection of S
- Create a heap with elements of S

IDEA 1

One idea is to run `Insert` for each element of S into the heap. The runtime is $\Theta(n \lg n)$ in the worst case.

IDEA 2

If $S = [\_, e_1, e_2, \ldots, e_n]$, this already can represent a binary tree structure (with on constraints). Instead of starting with an empty list and copying elements into the new list one-by-one, I'll work directly with the new list and re-order the elements. I'll work from the bottom up, with all the leaves.

The leaves in my list correspond to **half** of my values of the array. For any value in the second half of the list, if I multiply the index by 2, I'm out of the list anyway.

Start with the furthest node that isn't a leaf, at index $i = \lfloor \frac{n}{2} \rfloor$, and work backwards. The roots of its subtrees are all correct heaps. Run `Heapify` on that node, and for each index.

> For each index in order from $\lfloor \frac{n}{2} \rfloor$ to 1, call `Heapify`.

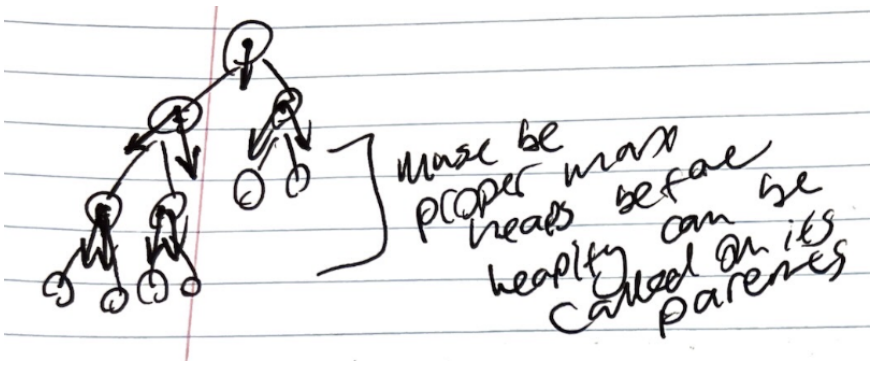

Figure 7: The idea of building a heap

2.7.1 Running Time of Building a Heap, Idea 2

In a complete binary tree with n nodes (assuming n is a power of 2):

- $\frac{n}{2}$ of them are leaves. We don't need to do work for any of them

- $\frac{n}{4}$ of them have height 1, and requires a max. of 1 level swap
- $\frac{n}{8}$ of them have height 2, and this requires a max. of 2 level swaps
- $\vdots$

As I go up the tree, the number of nodes I need to do go down quickly.

The total work is $n\sum \frac{i}{2^i} = \Theta(n)$.

2.8 Heap Sort

Suppose I have this list:

$$[4,\ 3,\ 7,\ 1,\ 8,\ 5],\ \text{len} = 6$$

I want to sort this list using heapsort. The first thing that heapsort does:

1. Build a max heap from the array: $\Theta(n)$
   a. $\rightarrow [4,\ \mathbf{8},\ 7,\ 1,\ \mathbf{3},\ 5] \rightarrow [\mathbf{8},\ 4,\ 7,\ 1,\ 3,\ 5]$
2. Call `EXTRACT-MAX` $n-1$ times: $\Theta\left(n\log(n)\right)$
   a. $\rightarrow [\mathbf{7},\ 4,\ \mathbf{5},\ 1,\ 3,\ |\ 8]$. All items on the right of the $|$ is not part of the heap, as signaled by the length of the heap, which **must** be tracked. `INSERT` overwrites the junk area, but I won't be running that.
   b. $\rightarrow [\mathbf{5},\ 4,\ \mathbf{3},\ 1,\ |\ 7,\ 8]$
   c. $\rightarrow \vdots$

After the $n-1^{\text{st}}$ `EXTRACT-MAX` call, I would already end up with a sorted list. The time complexity is $\Theta\left(n\log(n)\right)$

3 Dictionaries

A dictionary is a **set** where each element has a **unique** key: `x.key`. This means objects held in the dictionary must have its key.

Two objects can be the same for everything except its key, and a set can contain both, as they won't be equal.

The operations:

- `SEARCH(S, k)`: return element $x \in S$ with `x.key == k`. Or `NIL` if I can't find any (programming language agnostic `NULL`).
- `INSERT(S, x)`: Add x to S. If S contains element y with `y.key == x.key`, remove the old element y and replace it with x.
  - This means I have to actively look for duplicate keys to avoid duplicates.
- `DELETE(S, x)`: remove element x from S.
  - NOTE: `DELETE(S, SEARCH(S, k))` is used if you want to delete something based on a key. This implementation of `DELETE` prevents issues of needing to find a key given an element.

3.1 Data Structures / Implementations

Note that I must be able to access the length of the array in constant time. If something is sorted, the keys must be comparable. Linked lists are doubly linked lists.

Structure / OP	`SEARCH(k)`	`INSERT(x)`	`DELETE(x)`
Unsorted array	$\Theta(n)$	$\Theta(n)$ – inspect keys	$\Theta(1)$ (If I know the index. Memory leak?)
Sorted array	$\Theta(\lg(n))$	$\Theta(n)$ – array issues	$\Theta(n)$

Structure / OP	`SEARCH(k)`	`INSERT(x)`	`DELETE(x)`
Unsorted linked list	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Sorted linked list	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$
Direct access table (memory hog)	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Hash tables	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Binary search trees	Height $\mathcal{O}(n)$	Height $\mathcal{O}(n)$	Height $\mathcal{O}(n)$
Balanced search trees	$\Theta(\lg(n))$	$\Theta(\lg(n))$	$\Theta(\lg(n))$

3.2 Binary Search Trees

Our implementation is recursive, and we're going to write them in a style, as it makes it a lot easier to talk about balancing them.

ASSUMPTIONS:

- BST nodes store the following:
  - item
    - key (required to implement dictionary)
    - value
  - left
  - right
- Dictionary only stores the root: `S.root`

The operations go as follows:

- `INSERT(S, x):`

- `S.root = BST_INSERT(S.root, x)`. This is a recursive helper and returns the root of the resulting tree.

Helper functions:

```
def BST_INSERT(root, x):
    """Add x to the tree starting at root.
    Return the root of the tree afterwards.
    """
    if root == NIL:
        root = BSTNode(x)
    # ensure recursive cases happen first
    elif x.key < root.item.key:  # INSERT LEFT
        root.left = BST_INSERT(root.left, x)
    elif x.key > root.item.key:  # INSERT RIGHT
        root.right = BST_INSERT(root.right, x)
    else:  # x.key == root.item.key
        # remove the old item to prevent
        # key duplicates
        root.item = x
    return root
```

3.2.1 Runtime For Binary Trees

For a BST, worst case is $\Theta(n)$. When I have a balanced BST, the runtime returns to $\log(n)$. This calls for balanced search trees.

3.3 AVL Trees (Balanced Binary Trees)

An AVL tree is a balanced binary tree.

How do we balance trees, and how do we fix things when things go out of balanced?

Something we do to a structure of a binary tree by rearrange a few references. We can move things around in a BST of it such that the tree structure changes, but the ordering of the values do not change.

We can use tree rotations to do this:

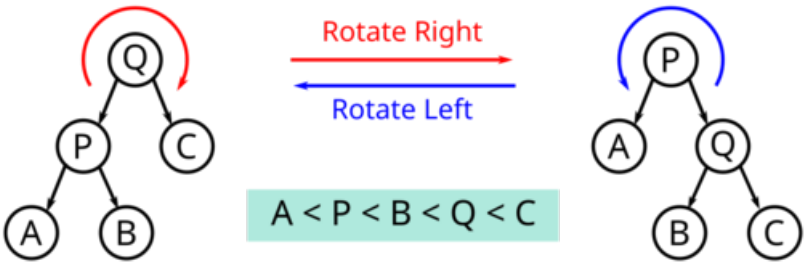

Figure 8: Tree rotations. Image from Wikipedia, see Binary Tree Rotations

When do we use this?

- The BST search algorithm doesn't need this.

We're not going to try to keep the tree perfectly always balanced. Doing so will require to move a lot of nodes at once. We need the tree to be roughly balanced.

Empty subtrees have a height of -1. They refer to `NIL`.

Completely balanced means

- All subtrees have the same height

Approximately balanced means:

- For each node:
  - `Height(left subtree)== Height(right subtree)` ± 1
  - Allow an error of 1

Property: Binary trees that are approximately balanced have height of $\Theta(\log(n))$.

So how are we going to insert:

1. Insert like normal, ignoring the fact that the insertion needs to be balanced.
2. Starting at the insertion point and work up the tree, using rotations to fix balance where needed.
   a. If the left node is heavier, rotate right

b. If the right node is heavier, rotate left

c. To do this, nodes must track their height

Tree height must be kept updated during operations, and during rotations. This can be done in constant time per node on the path from the root to the point in the tree where we made the update.