02-613 Week 7

Algorithms and Advanced Data Structures

Aidan Jan

October 7, 2025

Amortized vs. Average

Recall that average refers to events in isolation and amortized refers to across the whole run. We have used this for runtime analysis multiple times:

- Tree traversal (BFS/DFS): It takes $|V| + |E|$ to run a traversal on a graph, but if we have a tree, we only run on a subset of edges. Therefore we don't pay the cost of running on the whole graph.
- Heap insert: It takes $O(\log(n))$ to add a node to a heap, but not every node will take that long. If we know the dataset, it can be lower.

Binary Increments

Suppose we are counting in binary. To go from 0 to 1, we flip 1 bit. From 1 to 2, we flip 2 bits. From 2 to 3, we flip 1 bit again. But from 3 to 4, we flip 3 bits.

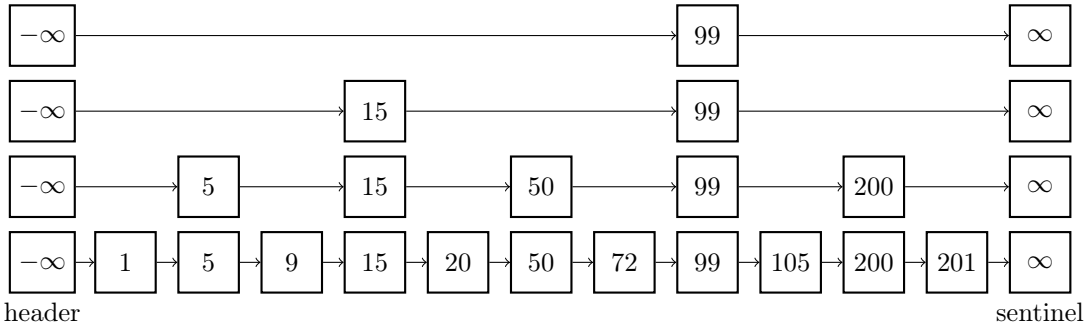
- In the worst case, notice that we flip k bits. However, we would not say that flipping bits in binary is $\mathrm{O}(k)$, because we *know* that not every bit is flipped every iteration.
- Claim: cost to increment from 1 to k is $2k$.
  - The idea is that we "prepay" $1 \to 0$ flips. We know that whenever we flip a bit from 0 to 1, we will eventually flip it back from 1 to 0.
  - On every increment, we flip exactly one bit from $0 \to 1$. All others are $1 \to 0$, which have already been "paid" for.
  - From this, we can show that each operation costs 2, and not $\log(k)$, even though $\log(k)$ bits are flipped in the worst case.
  - This improves our runtime estimate to $\mathrm{O}(k)$ instead of $\mathrm{O}(k \log k)$

Improving Linked Lists

Why do we ever use linked lists if all its operations are slower than arrays? For example. get, length, and replace in an array are all $\mathrm{O}(1)$, but linked list are all $\mathrm{O}(n)$. A linked list is just slower to use.

- This is where the perfect skip list comes in!
- Start with a linked list
- Promote half the nodes to a higher "level"
- Keeps a "head", "sentinel" to mark front and back
- Each node is on between 1 + $\mathrm{O}(\log n)$ levels

- Higher levels skip nodes



- We want to keep the sorted nature of this structure whenever nodes are inserted.

Search

- Sorted allows us to search nodes in $O(\log n)$ time. For example, to find 201, we start from the header, $-\infty$, then compare to 99.
  - $201 > 99$ (level 3), so we keep checking. $201 < \infty$ so we move down a level (to level 2).
  - $201 < \infty$ (still) so we move down another level
  - $201 > 200$ so we move across to the 200 (level 1) node.
  - $202 = 201$, so we move across. We're done!

Insertion

To insert, we first find the spot to insert on the first level, which contains every node. Then, we randomly promote a levels on nodes.

- Literally, insert node at level 1. `while (coin_flip == True):  insert node to next level`. This process repeats infinitely.

Analysis

We always enter the structure from the highest level. For the purposes of runtime analysis we will consider the path taken to get to a node.

- Consider the backwards walk from any element. We always want to go as high as possible to retrace our steps.
- The probability we go up on a step is $\frac{1}{2}$. This means we can either go left or up, with a 50% chance at either one.
- To get to the $log(n)$-th level, it will take $2 \log n$ steps.
- With high probability, the average search time is $O(\log n)$

Splay Trees

The idea:

- Modify a BST
- More frequently accessed nodes are placed near the top
- Simple to implement

- Amortized O($\log n$) lookup

We have the function `splay(T, k)`, which moves a given node k to the root (with some asterisks). Basically,

```
Splay(T, k):
    if k is in T:
        move k to the root
    else:
        either the inorder predecessor or ancestor moves to the root.
```

To understand how the splay function would work, consider a binary search tree with J as the root. Suppose we have a node k we would like to splay to the top.

- To do this, if $k > J$, then set k as the root, take the right subtree of J and make it the right subtree of K, and append J (along with the left subtree) to be the left child of k.
- In the opposite case, $k < J$, we do the similar thing with making k the root, appending the left subtree of J to be the left subtree of k, and appending J and the right subtree to the right child of k.
- The result is a valid binary search tree with the desired node k at the top. However, it is not necessarily balanced.

Find Item

To find the item, we splay the item we want to the top, and check if it is the root.

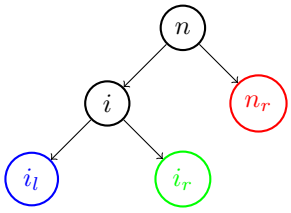
Concatenate Two Trees

Suppose we have T_1 and T_2, and we would like to combine them in the splay tree. How do we do that?

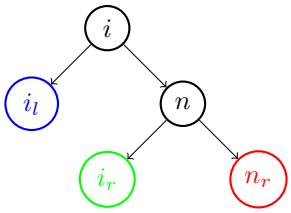
- We want to note that the new root must be either the smallest node of the larger tree, or the largest node of the smaller tree. This is because binary trees must have everything on the left smaller than the root, and everything on the right larger than the root.
- To do this, we simply run `Splay(T1, inf)` on the smaller tree. Since the largest node in the tree is the closest to infinity out of everything else in the tree, it will propagate the largest node to the top.
- Now, make that node the root of both trees, and use the rest of the smaller tree as the left child. The other tree becomes the right child.

So How does Splay() Work?

We use a concept called "rotations".

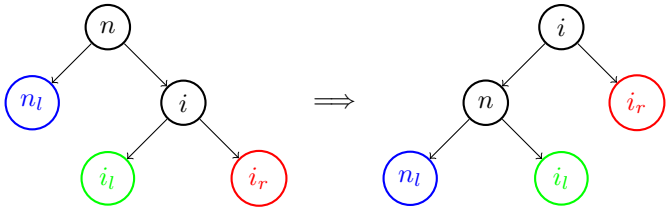


Consider the following tree structure, where n is the root, i is the node we want to be the root, and i_l, i_r, and n_r are the left and right subtrees of i and the right subtree of n, respectively. To "rotate" i to the root, we make the structure the following:



Basically, we bubbled i to the top, and since it was smaller than n before, $n > i$, so n becomes the right node.

In the opposite case, where i was the right subchild to begin with, we get:

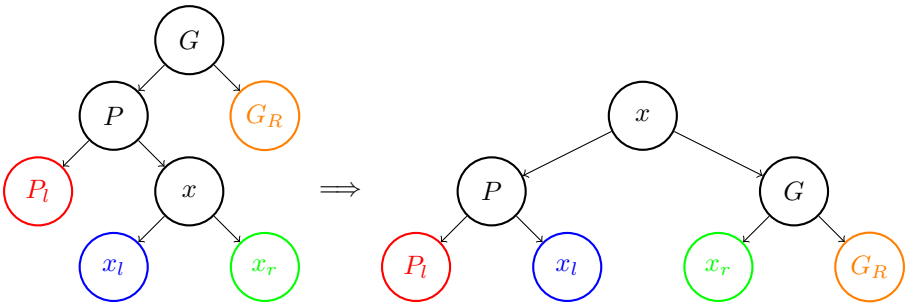


A single rotation is called a **Zig**.

ZigZag

A zig does not affect grandparents. However, a zigzag does. This operation is composed of basically two Zig operations

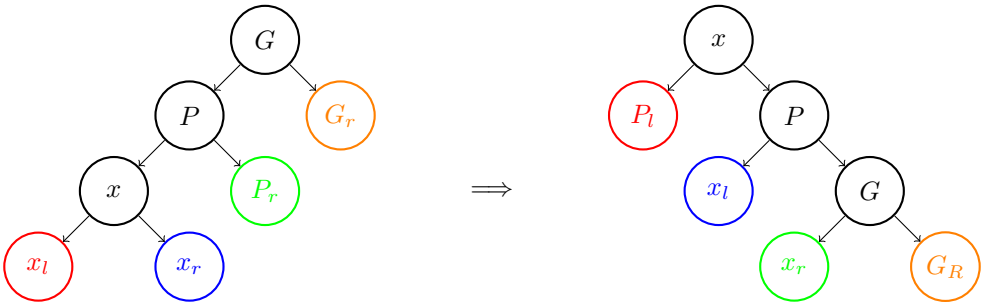
- The first is on node (P), where (x) moves up and (P) becomes (x)'s left child.
- The second is on node (G), where (P) moves up and (G) becomes (P)'s right child.



ZigZig

This operation is also composed of two Zig operations:

- The first is on node (G), where (P) moves up and (G) becomes (P)'s right child, resulting in a fully balanced tree.
- The second is on node (x), where (x) moves up and (P) becomes (x)'s right child.



Cost of Splay

What we really care about in runtime of the splay operation is the change of rank.

- Assign every node in the tree with a weight, where weight is defined as the number of nodes in the subtree rooted at that node. (e.g., all leaf nodes have weight 0)
- The rank of a node is defined as the floor of the log of its weight ($\text{rank}(u) := \lfloor \log \text{weight}(u) \rfloor$), and represents approximately how high up it is in the tree.

Theorem: It costs at most $3\lfloor \log n \rfloor + 1$ new dollars to splay and keep the money invariant.

- Each operation may be more expensive, but it can use savings.

Assume that we have the following costs for the varying splay operations:

- Zig = $3(r^1(x) - r^0(x)) + 1$
- ZigZag = $3(r^1(x) - r^0(x))$
- ZigZig = $3(r^1(x) - r^0(x))$

In this case, r^1 refers to the rank of the parent, and r^0 refers to the rank of the child. Why is this?

$$\begin{aligned} & 3(r^k(x) - r^0(x)) + 1 \\ & \leq 3r^k(x) + 1 \\ & \leq 3\lfloor \log(n) \rfloor + 1 \end{aligned}$$

- In the worst case, $r^0(x)$ is a leaf node, so the rank is 0.
- Also in the worst case, $r^k(x)$ is the root node, so the rank is $\log(n)$.

In the case of splaying multiple ranks, we have to use ZigZags and ZigZigs, so our total cost of the operation would be:

$$3(r^1(x) - r^0(x)) + 3(r^2(x) - r^1(x)) + \cdots + 3(r^k(x) - r^{(k-1)}(x)) + 1$$

Where each term is to move subtrees up or down.

- Refer to the costs described above. The +1 at the end comes from the sole Zig operation at the end.

Now, how do we prove these costs are correct?

Proof

First, we need a few lemmas:
Lemma 1:

$$\text{cost(zig)} \leq 3(r^1(x) - r^0(x)) + 1$$

A zig needs $3(r^1(x) - r^0(x)) + 1$. This relies on the fact that $r^0(P(x)) = r^1(x)$. In other words, the total number of nodes below $P(x)$ in the original tree is equal to the number of nodes below x. The +1 at the end pays for the actual rotation to swap the two nodes.

We can write the operations for a zig as

$$[r^1(x) + r^1(P(x))] - [r^0(x) + r^0(P(x))]$$

where the left represents the number of dollars needed on x and $P(x)$, while the right represents the number of dollars we already have on x or $P(x)$. It is worth noting that $r^1(x)$ on the left term cancels the $r(P(x))$ in the right term, since they are equal. Therefore, we can simplify:

$$\begin{aligned} & = r^1(P(x)) - r^0(x) \\ & \leq r^1(x) - r^0(x) \\ & \leq 3[r^1(x) - r^0(x)] \end{aligned}$$

Lemma 2:

$$\text{cost(zigzig)} \leq 3(r^1(x) - r^0(x))$$

A zigzag needs $r^1(R) - r^0(x)$ new dollars. R refers to the grandparent node, and S (which will be seen later) is the parent node we are doing the zigzag operation around. In other words, $G \mapsto R$, $P \mapsto S$ in the diagrams above.

Assume that $r^1(R) = r^1(S) \cdot \alpha$, where α is defined as $\alpha = r^1(x) = r^0(R) = r^0(S) = r^0(x)$. We know that

$$2^\alpha \leq w^1(R) \leq 2^{\alpha+1}$$
$$2^\alpha \leq w^1(S) \leq 2^{\alpha+1}$$

Here, α is some constant, and w is the weight of the node. However, we know that

$$w^1(x) \geq w^1(R) + w^1(S)$$

because at the end, the node x is at the top, and therefore its weight must be greater than that of both its children. Therefore,

$$w^1(x) \geq 2 \cdot 2^\alpha = 2^{\alpha+1} \Rightarrow r^1(x) = \alpha + 1$$

This is a contradiction since $r^1(x) := \alpha$.