CMSC 420 (Advanced Data Structures)

Instructor: Justin Wyss-Gallifent
Textbook: none
Type: #class
Topics: Data Structures Trees

Amortized Analysis

- Motivation: common for data structures to have:
  - Lots of cheap operations
  - Some expensive operations
  - Suppose we take a sequence of n operations (as bad as possible); what is avg per-op cost?
  - Examples
    - Ex: we randomly choose a binary tree with 3 nodes (all equally likely). What do we expect the height to be?
      - 4 cases with height 2, 1 case with height 1, so

$$\frac{4}{5}\cdot 2+\frac{1}{5}\cdot 1=\frac{9}{5}$$

        - This type of thinking is useful for average case analysis/counting structures
    - Ex: consider the below pseudocode

```
function sum(n)
        sum = 0
        for i = 1 to n
            sum = sum + 7
            if i is an integer power of 3
                sum = sum + i
            end if
        end for
        return sum
    end function
```

      - Can we find a closed form for the above?
        - For arbitrary n, we have

$$(7+1)+(7)+(7+3)+(7)+(7)+(7)+(7)+(7)+(7+9)+\cdots$$

          Separating this, we have

$$(7n)+(1+3+9+\cdots)$$

          where the last power of three is the largest power of 3 $\leq n$, or largest k such that $3^k\leq n$
        - This means $k\leq \log_3(n)$; as $k\in\mathbb{Z}$, we have $k=\lfloor \log_3 n\rfloor$
        - So, a closed formula for our sum is

$$\begin{aligned} 7n + \sum_{i=0}^{\lfloor \log_3 n \rfloor} 3^i &= 7n + \frac{3^{\lfloor \log_3 n + 1 \rfloor} - 1}{3 - 1} \\ &= 7n + \frac{3^{\lfloor \log_3 n \rfloor + 1} - 1}{2} \end{aligned}$$

- Ex: List deletion
  - Consider a 0-indexed list of length n; choose an index at random and delete that element. All elements to the right shift left by 1; each shift takes 0.2ms; on average, how long will a deletion take?
  - If we delete an element at index i, how many elts get shifted? $n - i - 1$
    - This takes $0.2(n - i - 1)$ time, so on average, the EV is

$$\sum_{i=0}^{n-1} \frac{1}{n} \cdot 0.2 \cdot (n - i - 1) = \frac{0.2}{n} \left[\sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i - \sum_{I=0}^{n-1} 1 \right]$$

- Ex: Suppose we have $m \geq 2$ and h represent the height of a tree with the following properties
  - All leaves are at the same level (tree is perfect)
  - Each node contains $m - 1$ keys
  - Each node except the leaves has m children
  - Each node except the leaves has m children
  - Q: in general, for an unknown h, m, how many keys are there? Make a table:

level	# nodes	# keys
0	1	$m - 1$
1	m	$m(m-1)$
2	m^2	$m^2(m-1)$
$\vdots$	$\vdots$	$\vdots$
h	m^h	$m^h(m-1)$

- Techniques for amortized analysis
  - Aggregate method
    - Given a situation, think of a sequence of n operations for which the total cost $C(n)$ is as bad as possible
    - Def: If the total cost of an operation is $C(n)$, the **amortized cost** is $AC(n) = \frac{C(n)}{n}$
    - Ex: suppose we have a data structure with the following:
      - Insertion + deletion each cost 1
      - When we have a multiple of 10 objects in the structure, there is an additional cost of 8
      - Suppose we perform n operations starting with an empty structure; to obtain worst-case, we would continuously insert, resulting in a total cost and amortized cost of

$$\begin{aligned} C(n) &= n + 8 \left\lfloor \frac{n}{10} \right\rfloor \\ AC(n) &= \frac{C(n)}{n} = \frac{n + 8 \left\lfloor \frac{n}{10} \right\rfloor}{n} \end{aligned}$$

      As $\lfloor x \rfloor \leq x$, we have

$$AC(n) \leq \frac{n + 8 \left(\frac{n}{10} \right)}{n} = 1 + \frac{8}{10} = \frac{9}{5}$$

      so $AC(n) \in \mathcal{O}(1)$
    - Ex: we allocate space for a stack, starting with 0 bytes. We can push/pop, each costing 1. If we overflow the allocated space we allocate more space and copy all the old stuff over, then push. Suppose the allocation + copy costs a value equal to the size of the new space; how do we reallocate?

- Option 1: Always allocate 1 more byte. In the worst-case, all we do is push, which will cost a

$$C(n) = (1+1) + (2+1) + (3+1) + \cdots + (n+1) = n + \frac{n(n+1)}{2} \in \mathcal{O}(n^2)$$

- Option 2: when we need to reallocate, double the space. In the worst-case, we have

$$C(n) = n + (1 + 2 + \cdots + 2^k)$$

for some k. Since we allocate to make space, we know $2^k \geq n$, so we need the smallest k such that $2^k \geq n$, or $k \geq \log_2 n$, which implies $k = \lceil \log_2 n \rceil$. Thus, we have

$$C(n) = n + \sum_{i=0}^{\lceil \log_2 n \rceil} 2^i = n + \left[2^{\lceil \log_2 n \rceil} - 1\right]$$

so

$$\begin{aligned} AC(n) = \frac{C(n)}{n} &= \frac{n + 2^{\lceil \log_2 n \rceil + 1} - 1}{n} \\ &\leq \frac{n + 2^{\log_2 n + 1 + 1} - 1}{n} \\ &\leq \frac{n + 4n - 1}{n} = \frac{5n-1}{n} = 5 - \frac{1}{n} \leq 5 \end{aligned}$$

- So $AC(n) \in \mathcal{O}(n)$

- More examples
  - Ex: we randomly choose a binary tree with 3 nodes (all equally likely). What do we expect the height to be?
    - 4 cases with height 2, 1 case with height 1, so

$$\frac{4}{5} \cdot 2 + \frac{1}{5} \cdot 1 = \frac{9}{5}$$

    - This type of thinking is useful for average case analysis/counting structures
  - Ex: consider the below pseudocode

```
function sum(n)
    sum = 0
    for i = 1 to n
        sum = sum + 7
        if i is an integer power of 3
            sum = sum + i
        end if
    end for
    return sum
end function
```

  - Can we find a closed form for the above?
    - For arbitrary n, we have

$$(7+1) + (7) + (7+3) + (7) + (7) + (7) + (7) + (7) + (7+9) + \cdots$$

Separating this, we have

$$(7n) + (1 + 3 + 9 + \cdots)$$

where the last power of three is the largest power of 3 $\leq n$, or largest k such that $3^k \leq n$

- This means $k \leq \log_3(n)$; as $k \in \mathbb{Z}$, we have $k = \lfloor \log_3 n \rfloor$
- So, a closed formula for our sum is

$$\begin{aligned} 7n + \sum_{i=0}^{\lfloor \log_3 n \rfloor} 3^i &= 7n + \frac{3^{\lfloor \log_3 n + 1 \rfloor} - 1}{3 - 1} \\ &= 7n + \frac{3^{\lfloor \log_3 n \rfloor + 1} - 1}{2} \end{aligned}$$

- Ex: List deletion
  - Consider a 0-indexed list of length n; choose an index at random and delete that element. All elements to the right shift left by 1; each shift takes 0.2ms; on average, how long will a deletion take?
  - If we delete an element at index i, how many elts get shifted? $n - i - 1$
    - This takes $0.2(n - i - 1)$ time, so on average, the EV is

$$\sum_{i=0}^{n-1} \frac{1}{n} \cdot 0.2 \cdot (n - i - 1) = \frac{0.2}{n} \left[\sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i - \sum_{I=0}^{n-1} 1 \right]$$

- Ex: Suppose we have $m \geq 2$ and h represent the height of a tree with the following properties
  - All leaves are at the same level (tree is perfect)
  - Each node contains $m - 1$ keys
  - Each node except the leaves has m children
  - Each node except the leaves has m children
  - Q: in general, for an unknown h, m, how many keys are there? Make a table:

level	# nodes	# keys
0	1	$m - 1$
1	m	$m(m - 1)$
2	m^2	$m^2(m - 1)$
$\vdots$	$\vdots$	$\vdots$
h	m^h	$m^h(m - 1)$

- Token (banker's) method (described in Justin's notes)
  - Typically used when worst-case situation can be divided into **runs** (segments) in which each run consists of a sequence of cheap operations ending with an expensive operation
  - Ex: Suppose we're looking at lunch costs for the week (a run); from Sunday t Friday you spend $8 per day and on Sat, you spend $29
    - On average how much did you spend per day? $\frac{8 \cdot 6 + 29}{7} = 11$ per day
    - This is an aggregate analysis
    - Rephrased example: every day you put β into the bank and withdraw as needed for lunch. What must be true of β to ensure enough? As we have 7 days and put β per day, we need $7\beta \geq 77 \geq 8 \cdot 6 + 29$, so $\beta \geq 11$
    - Instead of money, think of time; we've calculated the pre-operation cost
  - Ex: stack allocation; if we overflow our stack we double the space allocated
    - A run will start after a reallocation + push and will stop after the next reallocation + push; more rigorously, for some k
      - Run starts: we reallocated from $2k$ to $4k$ and there are $2k + 1$ elts on stack
      - Run stops: we reallocate from $4k$ to $8k$ and there are $4k + 1$ elts on stack
      - Total run cost: push cost + reallocation cost = $2k + 8k = 10k$; we want β such that $\beta(2k) \geq 10k$, so $\beta \geq 5$, so $\beta = \mathcal{O}(1)$

- Ex: stack allocation; when we reallocate, go to the next perfect square (e.g. $0 \to 1 \to 4 \to 9 \to 16 \to \cdots$), for some k
  - Run start: we reallocate k^2 to $(k+1)^2$ and (k^2+1) elts on the stack
  - Run stops: we reallocate $(k+1)^2$ to $(k+2)^2$ and $(k+1)^2+1$ elts on the stack
  - Total operations: $[(k+1)^2+1]-[k^2+1] = k^2+2k+2-k^2-1 = 2k+1$ (like the number of days in above ex)
  - Total run cost: push cost + reallocation cost = $2k+1+(k+2)^2 = k^2+6k+5$ (like the total meal cost in above ex)
  - So $\beta(2k+1) \geq k^2+6k+5$, implying $\beta \geq \frac{1}{2}k + \frac{11}{4} + \frac{9/4}{2k+1}$, which implies

$$\beta \geq \frac{1}{2}k + \frac{11}{4} + \frac{9}{4}$$
$$\beta \geq \frac{1}{2}k + \frac{20}{4}$$
$$\beta \geq \frac{1}{2}k + 5$$

  - What about this k? We must have $k \geq 1$ and as we're pushing n items we must have $(k+1)^2+1 \leq n$ which is $k \leq \sqrt{n-1}-1$, so together we have

$$1 \leq k \leq \sqrt{n-1}-1$$

  - If we have $\beta \geq \frac{1}{2}(\sqrt{n-1}-1)+5$ then we will be safe, but in addition we want β as small as possible so we let $\beta = \frac{1}{2}(\sqrt{n-1}-1)+5$, so $\beta = \mathcal{O}(\sqrt{n})$
- Potential method

Trees

- Motivation: why will we look at so many trees?
  - They work; if managed well, good runtimes for insertion/deletion/querying
  - "Easy" to understand
  - Some properties of trees
    - Binary search tree property
      - For any node, it's key is less than all keys in the right subtree and greater than all keys in the left subtree
    - Max heap property
      - For any node, its value is greater than or equal to all the values of its children
    - Tree height: the length is the length of the longest path from root to the leaf (path length is equal to the number of edges)
    - Tree levels
      - Typically 0-indexed from the root down
      - There are rare, weird alternative defns of tree levels (see AA trees for example)
    - Traversal types
      - Breadth-first
      - Depth-first
      - Pre-order
      - Post-order
      - In-order
    - Storing trees
    - Balanced trees: what could it mean for a tree to be balanced?
      - Weight (size) balanced

        - $\text{weight}(L.\text{subtree}) = \text{weight}(R.\text{subtree})$
    - Height balanced
        - For every node, we have $\text{height}(L.\text{subtree}) = \text{height}(R.\text{subtree})$
        - AVL trees/scapegoat trees

Binary Search Trees

- Operations: search, insertion, delete
    - Deletion: $\text{search}(x)$
        - Steps: start at the root, go left or right depending on how x compares to the node's key, and keep going until we find x (or fall out and error)
        - Suppose we have a BST with n nodes
            - Best case search is $\Theta(1)$ and worst-case is $\Theta(n)$
            - Average case is tricky, because for a tree with n nodes, it could have been created with just inserts or with inserts and deletions
                - This matters because deletion and associated choices can affect this result
    - Insertion: $\text{insert}(x)$
        - Steps:
            - "Search" for x
            - When we reach bottom of tree, search concludes
            - Time complexity
                - Best-case, worst case are $\Theta(1)$, $\Theta(n)$, respectively
    - Deletion: $\text{delete}(x)$
        - Steps:
            - Search for x
            - If it is a leaf, drop if off
            - If it has one child, we remove the node and promote the child
            - If it has two children, we have two options
                - In-order predecessor, the largest node/key smaller than x
                - In-order successor, the smallest node/key larger than x
                - Pick one of the above, replace it with x, and recursively delete that node/key (this replacement is easy to delete because it will have at worst one child)
    - Average case of insertion/deletion/search depends on the average height of the tree
        - For a BST with n nodes, we have
            - (a) Max height: $\Theta(n)$
            - (b) Min height: $\Theta(\log_2 n)$ if the BST is perfect
            - (c) Average height; to find this, we can
                - (a) Look at all binary trees with n nodes and take the average height
                    - This has been proven to be $\Theta(\log_2 n)$
                - (b) Look at all possible constructions of BST using insertion/deletion, which ends up with n nodes
                    - If we always use the same option between in-order predecessor and in-order successor for deletion, then it is $\Theta(\sqrt{n})$ (worse than $\Theta(\log_2 n)$
                    - If we choose randomly between in-order predecessor and in-order successor, the data suggests $\Theta(\log_2 n)$ again (has not been proved)

AVL Trees

- Main issue with binary search trees

- Insert, search, delete operations are worst-case $\mathcal{O}(n)$ because BSTs can become unbalanced
- Can we tweak our BST so it always stays balanced?

- Definitions
  - If x is a node in a binary search tree, define

$$\text{balance}(x) = \text{height}(x.\text{rightsubtree}) - \text{height}(x.\text{leftsubtree})$$

    - Recall $\text{height}(\text{emptytree}) = -1$
  - A binary search tree satisfies the **AVL balance condition** (is an AVL tree) if for all nodes x, $\text{balance}(x) \in \{-1, 0, 1\}$
- Height bound: if an AVL tree has height $\mathcal{O}(\log n)$, the operations insert/deletion/search should be faster
  - Thm: suppose an AVL tree has n nodes and height h. Then $h \in \mathcal{O}(\log n)$.
    - Proof: Define $N(h)$ to be the minimum number of nodes in an AVL tree with height h (e.g. $N(0) = 1$). Now, for every h, we have $N(h-1) > N(h-2)$ because the $h-1$ term is taller/higher, so it has more nodes. Also, $N(h) = 1 + N(h-1) + N(h-2)$ because the tree is an AVL tree (but there are as few nodes as possible on each side). From here, we have

$$\begin{aligned} N(h) &= 1 + N(h-1) + N(h-2) \\ N(h) &> 1 + N(h-2) + N(h-2) \\ \rightarrow N(h) &> 2n(h-2) \end{aligned}$$

      Now, we have $N(h) > 2N(h-2) > 2^2N(h-4) > 2^3N(h-6) > \dots$
      We now have two cases:

      - Case 1: h is even. This ends at $N(h) > 2^{h/2}N(0)$. Since $N(h)$ is the minimum number of nodes in an AVL tree with height h for *any* AVL tree of height h, we have $n \geq N(h)$, so $n > 2^{h/2}N(0)$. As $N(0) = 1$, we have $n > 2^{h/2}$, implying $h/2 < \log_2 n$, or $h < 2\log_2 n$.
      - Case 2: h is odd. This is similar to the above case>

      Thus, we have $h \in \mathcal{O}(\log_2 n)$, as desired.
  - Note that inserts/deletions could ruin the balance condition; this brings us to the concept of **rotations**
    - Rotations
      - Left rotation
      - Right rotation (mirror image of a left rotation)
      - Left-right rotation
        - Step 1: Left rotation at left child
        - Step 2: Right rotation on parent
      - Right-left rotation
        - Step 1: Right rotation on right child
        - Step 2: Left rotation on parent

- Operations
  - Insertion
    - Step 1: First insert as with a vanilla BST
    - Step 2: Nodes along path from newly inserted leaf node up to root may be unbalanced
      - Traverse up to tree looking for node x with $\text{balance}(x) \notin \{-1, 0, 1\}$
      - If no unbalanced node is found, we are done
    - Step 3: If unbalanced node is found
      - Cases
        - Unbalance arises in left subtree (left-heavy)
          - Left-left heavy: do a right rotation
          - Left-right heavy: do a left-right rotation

        - Unbalanced arises in right subtree (right-heavy)
            - Right-right heavy: do a left rotation
            - Right-left heavy: do a right-left rotation
    - Things to note
        - In all four of the above cases, the balance of the root is 0
        - Height of resulting subtree is same as it was before insertion (all nodes above subtree) will be balanced
- Worst-case time complexity
    - Rebalancing is $\mathcal{O}(1)$, but we may need $\mathcal{O}(\log_2 n)$ height inspections to find an unbalanced node, so total worst-case time complexity of insertion is $\mathcal{O}(\log_2 n)$
- Deletion
    - Step 1: delete as with a vanilla binary search tree (including replacement). Then go to parent of the deleted node (or replacement node) and similar to insertion, go up looking for unbalanced nodes
    - Step 2: If an unbalanced node is found
        - Cases
            - Right-right heavy (and *possibly* right-left heavy): left rotation will fix
            - Right-left heavy (not right-right heavy): right-left rotation will fix
            - Left-left heavy: symmetric to right-right heavy
            - Left-right heavy: symmetric to right-lefty heavy
    - Worst-case time complexity
        - Steps
            - Step 1: Finding the node and possibly a replacement takes $\Theta(\log_2 n)$ because of the height of the tree
            - Step 2: Deletion (chop, splice, promote) is $\Theta(1)$
            - Step 3: Checking every node back to the root is $\Theta(\log_2 n)$ nodes
            - Step 4: Each fix (via a rotation) is $\Theta(1)$
        - Total worst-case time complexity is $\Theta(\log_2 n)$

B-Trees

- A **multiway search tree** is a generalization of a BST: if a node has keys $a_1 < a_2 < \ldots < a_k$, the children are roots of subtrees $A_1, \ldots, A_{k+1}$ with keys satisfying $A_1 < a_1 < A_2 < a_2 < \ldots < a_k < A_{k+1}$
- A **B-tree** of order $m \geq 3$ is a multiway search tree (MST) with the following properties:
    - It is **perfect** (all leaves are at the same level)
    - The root node has between 1 and $m-1$ keys; if the root node has children then $c = 1 + k$, so the number of children c is between 2 and m
        - Non-root nodes have between $\lceil \frac{m}{2} \rceil - 1$ and $m - 1$ keys
            - If the non-root has children than $c = 1 + k$, so the number of children is between $\lceil \frac{m}{2} \rceil$ and m
    - Note: we allow the root to have down to 1 key because we need to allow (for example) a B-tree with just 1 key
    - Note: we do not allow all nodes to have down to 1 key because it would permit the B-tree to degenerate to a vanilla BST, defeating the purpose of the B-tree
- Height theorem: Suppose a B-tree has order $m \geq 3$ and height h;. Then, the following properties hold:
    - (a) The minimum number of keys in the tree is $2\lceil \frac{m}{2} \rceil^h - 1$
    - (b) In general, $k \geq 2\lceil \frac{m}{2} \rceil^h - 1$
    - (c) $h = \mathcal{O}(\log k)$
    - Proof
        - To prove part (a), we will build a tree with the minimum number of keys via a table:

level	# nodes	# keys
0	1	1
1	2	$2(\lceil m/2 \rceil - 1)$
2	$\lceil m/2 \rceil$	$2\lceil m/2 \rceil(\lceil m/2 \rceil - 1)$
3	$2\lceil m/2 \rceil^2$	$2\lceil m/2 \rceil^2(\lceil m/2 \rceil - 1)$
$\vdots$	$\vdots$	$\vdots$
h	$2\lceil m/2 \rceil^{h-1}$	$2\lceil m/2 \rceil^{h-1}(\lceil m/2 \rceil - 1)$

So the total number of keys is

$$1 + \sum_{i=0}^{h-1} 2\lceil m/2 \rceil^i (\lceil m/2 \rceil - 1)$$

This is a geometric series which sums to $2\lceil m/2 \rceil^h - 1$, proving part (a). Part (b) follows directly from part (a). Using this result, we can obtain $h \leq \log_{\lceil m/2 \rceil} \left(\frac{k+1}{2} \right)$, proving $h = \mathcal{O}(\log k)$, as desired.

- Remark: we've shown that for a given m, h that $2\lceil m/2 \rceil^h - 1 \leq k \leq m^{h+1} - 1$, which says that given m, h, k is in that range. This *does not imply* that for each k in this range, which
- Example: suppose $m = 7$
  - If $h = 4$ then the minimum number of keys is $2 \left\lceil \frac{7}{2} \right\rceil 4 - 1 = 161$
  - If $k = 40000$ then $h \leq \log \lceil 9/2 \rceil \left(\frac{40001}{2} \right) \approx 9.02$ so $h \leq 9$

- Advantages of using B-trees
  - Less balancing required following insertions/deletions
  - Easier to do rang queries
- Restructuring
  - An **overfull node** is a node with $m + 1$ children (m keys)
  - An **underfull node** is a node with $\lceil m/2 \rceil - 1$ children and $\lceil m/2 \rceil - 2$ keys
  - Rotation operations
    - Key rotations: required when a node is over/underfull but there are extra keys in an adjacent sibling that we can use to restructure the tree

Scapegoat Trees

- Scapegoat trees are interesting because they involve several thought-provoking concepts
  - Time is analyzed via amortized analysis
  - Tree + subtree rebuilding
  - Weight-balancing
  - Balancing with insertion/deletion is very different
  - Very little additional info to store
- Balancing
  - Def: For a node x in a tree define $\text{size}(x)$ to be the number of nodes in the subtree rooted at x
  - Some ideas
    - For each node x we might require that $\text{size}(x.\text{left}) = \text{size}(x.\text{right})$ (this is too rigid)
    - For each node x, define $\text{size}(x.\text{left}) = \text{size}(x.\text{right})$, or equivalently $\frac{\text{size}(x.\text{child})}{\text{size}(x)} \approx \frac{1}{2}$
      - But $1/2$ is too rigid and defining an approximation is difficult (we need some formality)
    - Def: Let α be a real number with $\frac{1}{2} < \alpha < 1$. Define a **scapegoat** to be a node x with $\frac{\text{size}(x.\text{child})}{\text{size}(x)} > \alpha$
      - Typically, $\alpha = \frac{2}{3}$ is used, so a scapegoat has $\frac{\text{size}(x.\text{child})}{\text{size}(x)} > \frac{2}{3}$
      - We might think that a scapegoat tree is a tree with no scapegoats, but that is not where this is going
- A **scapegoat** is someone who is blamed for something they did not do (e.g. they are a sacrifice); a **scapegoat tree** is a binary tree which is only modified if it gets "badly unbalanced"

- Operations
  - Some formality
    - We will store only two things other than the tree
      - n, which is the number of nodes/keys
      - m, which is the maximum number of nodes which have been in the tree since it was created or since the last total rebuild due to deletion
      - Ex: start with an empty tree, so $n = 0$ and $m = 0$. Then, do the following operations:
        - Insert 17 keys, making $n = 17$ and $m = 17$
        - Delete 3 keys, making $n = 14$ and $m = 17$ (assume no rebuild)
        - Insert 1 key to obtain $n = 15$ and $m = 17$
    - Goals:
      - 1. Height of the tree should be $\lfloor \log_{3/2} n \rfloor + 1$ or less
      - 2. The tree should be "balanced" whenever possible
  - Sub(tree) rebuilding
    - Suppose we have a tree or subtree; we can rebuild it as follows:
      - First do an in-order traversal to get the keys in a list in order
      - If the list has length k, pick the entry with index $\lfloor k/2 \rfloor$ that becomes the new root
        - Then, all smaller keys go left via recursion
        - All larger keys go right via recursion
  - Insertion
    - Informal def: Insert as in a standard BST; if the insertion node is too deep (how deep? We will revisit this) we go back up the tree until we find a scapegoat (why must a scapegoat exist?) and we rebuild the subtree (what does this mean?) rooted at that scapegoat
    - Formal def: Insert as with a standard BST. Let d be the depth of the inserted node. If $d > \log_{3/2} n$, then go up the tree until we find a scapegoat and rebuild the subtree rooted at the scapegoat. We choose $3/2$ as the base of the logarithm because $1/\alpha = 3/2$
  - Deletion
    - Informal def: If we delete too many nodes (how many?) without inserting to compensate, we panic and rebuild the entire tree
    - Formal def: Delete as with a standard BST. If $n < \frac{2}{3}m$, rebuild the entire tree and set $m = n$
- Important results
  - Thm: Rebuilding a tree with j nodes takes time $\mathcal{O}(j)$
    - Proof: The inorder traversal via recursion can write to an array in $\mathcal{O}(j)$ time. As rebuilding is a recursive divide-and-conquer algorithm, the entire process takes time $\mathcal{O}(j)$
  - Thm: A rebuilt tree with j nodes will have height equal to $\lfloor \log j \rfloor$
    - Proof: Done by strong induction on j and is left to the reader
  - Scapegoat existence; if the depth of the inserted node is greater than $\log_{3/2} n$, then it has an ancestor which is a scapegoat
    - Thm: For a BST with n nodes suppose p is a freshly inserted node satisfying $\text{depth}(p) > \log_{3/2} n$. Then, an ancestor of p is a scapegoat.
      - Proof: Proceed by contradiction; suppose for every note u from the root to p we have $\frac{\text{size}(u.\text{childonpath})}{\text{size}(u)} \leq \frac{2}{3}$
        Following the path from the root r to node p, we have the following:

$$\begin{aligned} n &= \text{size}(r) \\ &\geq \frac{3}{2}\text{size}(r.\,\text{childonpath}) \\ &\geq \left(\frac{3}{2}\right)^2 \text{size}(r.\,\text{childonpath.childonpath}) \\ &\geq \dots \\ &\geq \left(\frac{3}{2}\right)^{\text{depth}(p)} \text{size}(p) = \left(\frac{3}{2}\right)^{\text{depth}(p)} \end{aligned}$$

So, $n \geq \left(\frac{3}{2}\right)^{\text{depth}(p)}$, which implies $\text{depth}(p) \leq \log_{3/2} n$

- Keeping track of size: size calculations do not slow processes down because they are in $\mathcal{O}$ of the size of a subtree

- Height theorems
  - Thm: The height of a scapegoat tree with $n \geq 1$ satisfies $h \leq \lfloor \log_{3/2} n \rfloor + 1$
    - Proof: Sketch in Justin's notes (more rigorous proof in Galperin/Rivest).
    - Ideas from proof:
      - A tree is **height-balanced** if $h \leq \lfloor \log_{3/2} n \rfloor$ and **loosely height-balanced** if $h \leq \lfloor \log_{3/2} n \rfloor + 1$ (an empty tree is height-balanced by default)
  - Corollary: The height of a scapegoat tree satisfies $h(n) = \mathcal{O}(\log n)$
    - Proof: Follows immediately from previous theorem.

- Time complexity
  - Search worst-case: as $\mathcal{O}(\log n)$, search is also worst-case $\mathcal{O}(\log n)$
  - Insertion/deletion: $\mathcal{O}(n)$
  - Amortized analysis
    - While insertion/deletion is worst-case $\mathcal{O}(n)$, such operations do not happen repeatedly (e.g. we will never do a rebuild immediately after a rebuild)
    - Thm: Starting with an empty tree, any sequence of k insertions/deletions has an amortized cost of $\mathcal{O}(\log k)$ per operation (including rebuilding)
    - See Justin's notes for a more detailed description

Splay Trees

- Motivation: we want an efficient self-balancing tree that easily gives access to nodes we need more access to
- Splay operations
  - **Zig**: when target node x is left or right child of p, do a right or left rotation at p, respectively
  - **Zag**: when target node x is left-left or right-right child of root g, do a right-right or left-left rotation at g, respectively
  - **Zig-Zag**: when target node x is left-right or right-left child of root g, rotate right at parent p and left at g in the first case or left at parent p and right at g
- Implementing splay
  - We **splay** a node when that node is the not the root of the tree and we wish to transport it to the root; we have the following four cases:
    - (a) If x is the root, we are done
    - (b) If x is a left-left child or a right-right child, Zig-Zig
    - (c) If x is a right-left child or a left-right child, Zig-Zag
    - (d) If none of the above apply, go back to (a)
- Tree operations
  - Search
    - Call splay on target key; if key exists, it will be at root (otherwise, throw an error)
    - Even if key is not at root, tree will be cleaner (more balanced and reorganized) after process

    - Insert
        - Algorithm 1: Described well in Justin's notes
        - Algorithm 2: Insert as with a vanilla BST
    - Delete
        - Algorithm 1: Described well in Justin's notes
        - Algorithm 2: Delete as with a standard BST and splay parent of that key to root
- Time complexity
    - Amortized cost of a single operation is $\mathcal{O}(\log n)$ because splay trees tend to be balanced
    - Worst-case is $\mathcal{O}(n)$, since there is technically nothing stopping the splay tree from turning into a linked list

KD Trees

- Motivation: we want to store keys in an n-dimensional space
- Cycle splitting
    - Insert using **cycle splitting** (for 2 dimensions, alternate between x and y coordinates to insert)
- Deletion
    - Must use inorder successor since we decided equal coordinates go right
    - If u (the node to be deleted) is a leaf, delete it and we are done
    - Otherwise, if u has a right subtree, let α be the coordinate that u splits on
        - Find a replacement node r in $u.\text{right}$ for which r_α (the α coordinate of r is minimal)
        - Copy r's point α to u (overwriting u's point)
        - Recursively call delete on the old r
    - If u does not have a right subtree, it has a left subtree, let α be the coordinate that u splits on
        - Find a replacement node l in $u.\text{left}$ for which l_α (the α coordinate of l) is minimal
        - Copy l's point to u (overwriting u's point) and move u's left subtree to become u's right subtree
        - Recursively call delete on the old l
        - Why does this work?
            - All other nodes in $u.\text{left}$ have α-coordinate greater than or equal to r_α, so when $u.\text{left}$ is moved to $u.\text{right}$, all nodes in $u.\text{right}$ are correctly positioned with respect to r_α

Extended KD-Trees

- Motivation: all data is at leaf nodes and internal nodes only contains coordinate to split on and splitting coordinate value (these are **splitting nodes**)
- Insertion
    - Note: we always go right on an equal coordinate
    - Insert at leaf; if leaf is overfull, split
    - Splitting leaf nodes
        - When we split, we create a parent splitting node that requires a splitting coordinate and value
        - Two possible approaches to splitting: **cycle split** or **spread split**

Character Tries

-