CMSC420 Cheatsheet

SUM FORMULAS

1. $\sum_{i=1}^{n} 1 = n$
2. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
3. $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$
4. $\sum_{i=0}^{n} r^i = \frac{r^{n+1}-1}{r-1}$ for $|r| < 1$
5. $\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$
6. $\sum_{i=0}^{n} i2^i = (n-1)2^{n+1} + 2$

AMORTIZED ANALYSIS

Things to add?

1. Why do we even do this? The goal is to obtain worst case per-operation cost
2. Token Method (a.k.a Bankers Method): Reallocation problems.

Complexity Classes

1. $o(n)$: $f < g$
   $f(n) \in o(g(n))$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$
   f grows **slower** *than g.*
2. $\mathcal{O}(n)$: $f \leq g$
   $f(n) \in \mathcal{O}(g(n))$ if $\exists C > 0, \exists n_0 \geq 1$ such that $\forall n \geq n_0, f(n) \leq Cg(n)$.
   Asymptotic upper bound.
3. $\Theta(n)$: $f = g$
   $f(n) \in \Theta(g(n))$ if $\exists B > 0, C > 0, n_0 \geq 1$ such that $\forall n \geq n_0, Bg(n) \leq f(n) \leq Cg(n)$
   Asymptotic upper and lower bound.
4. $\Omega(n)$: $f \geq g$
   $f(n) \in \Omega(g(n))$ if $\exists B > 0, \exists n_0 \geq 1$ such that $\forall n \geq n_0, f(n) \geq Bg(n)$
   Asymptotic lower bound.
5. $\omega(n)$: $f > g$
   $f(n) \in \omega(g(n))$ if $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$
   f grows **faster** *than g.*

Aggregate Method

Calculate the worst-case cost of n operations and then divide by n.

Token Method

Each operation has a cost of $\beta = AC(n)$ tokens. For cheap operations $\beta > 0$, for expensive operations, $\beta < 0$. Your goal is to find β.
For $\{x_1, \ldots, x_n\}$ operations, we want $(\beta - x_1) + \cdots + (\beta - x_n) = 0$.

AVL TREES

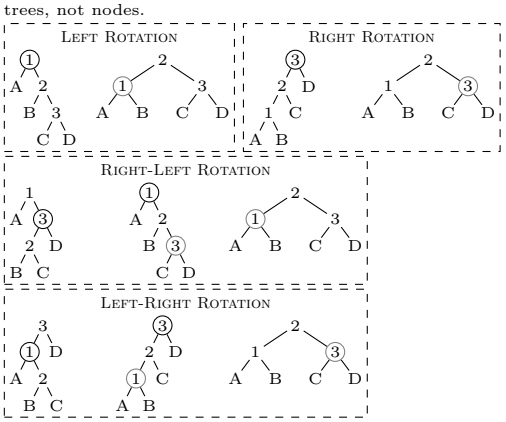
Balanced Binary Search Trees.

Height & Balance Factor

- For a node n with subtrees L and R we have $b(n) =$ `height`$(R) -$ `height`(L)
- A node n is *AVL balanced* if $b(n) \in \{-1, 0, 1\}$. All nodes must be AVL balanced.
- Height is $\Theta(\log n)$

Rotations

The node being rotated is circled. $A, B, C,$ and D are subtrees, not nodes.



Search

Same as BST.
Worst Case: $\mathcal{O}(\log n)$

Insert

Let n be the node to insert.

1. Insert n as with BST.
2. Recurse back up tree, looking for any node n' that is not AVL balanced. If found, rotate n' using one of the four rotations.

Worst Case: $\mathcal{O}(\log n)$

Delete

Let n be the node to delete.

1. Replace n with its inorder successor n'.
   Note: n' now either has only one, or zero subtrees.
2. If n' is a leaf, delete it.
3. If n' is not a leaf, promote its subtree.
4. Check if the parent of n' is now unbalanced and fix.

Worst Case: $\mathcal{O}(\log n)$

2-3 TREES

1. Always perfect.
2. Height $h \geq -1$, $\mathcal{O}(\log n)$ and $\mathcal{O}(\log k)$.
3. Empty if and only if $h = -1$.
4. B-Tree with $m = 3$

Parameters

1. h: Height of tree.
2. k: Number of nodes.
3. k: Number of keys.

Search

Same as BST.
Time Complexity: $\mathcal{O}(\log n)$.

Insert

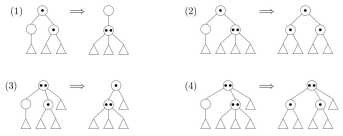
Let x be the key to insert.

1. Follow search path down to a leaf (always).
2. Add x to leaf.
   i. If this does not make the node overfull, done.
   ii. Otherwise, promote middle key up, left and right key become two nodes. Push the middle key into the node above. Possibly recurse this process.

Time Complexity: $\mathcal{O}(\log n)$.
Don't rotate ever when inserting! That's how it was invented

Delete

Do the BST thing until you reach a leaf, then remove the key. If you make a 0-key "hole", fix it like so:



Time Complexity: $\mathcal{O}(\log n)$.

B-TREES

Generalized version of 2-3 Tree.

1. Root has between 2 and m children, between 1 and $m-1$ keys.
2. All other nodes have between $\lceil m/2 \rceil$ and m children and $\lceil m/2 \rceil - 1$ and $m-1$ keys.
3. Always perfect.

Parameters

- m: Order of the B-Tree. Number of children.
- h: Height of tree.
- k: Number of keys.

Key Formulas

$2\left\lceil \frac{m}{2} \right\rceil^n - 1 \leq k \leq m^{h+1} - 1.$

Height Formulas

$\log_m(k+1) - 1 \leq h \leq \log_{\lceil m/2 \rceil}\left(\frac{k+1}{2}\right)$

Search

Let k be the key to search. Traverse each node until you find keys k_1, k_2 such that $k_1 < k < k_2$. Then, recurse to that child.
Worst Case: $\mathcal{O}(\log n)$ since we assume m is a constant. Recall that all log bases are the same to $\mathcal{O}$ notation.

Insert

We always insert into a leaf.
Let k be the key to insert.

1. Go through tree as if searching for k. If the node is not overfull, done.
2. If the node is overfull
   i. Check if either sibling has space. If so, rotate.
   ii. Otherwise, split the overfull node, promote the middle node, and recurse.

Worst Case: $\mathcal{O}(\log n)$ or $\mathcal{O}(\log k)$ for n nodes and k keys.

Delete

Let k be the key to delete.

1. Go through tree as if searching for k. If the node is not underfull, done.
2. If the node is underfull
   i. Check if either sibling can give. If so, rotate.
   ii. Otherwise, merge with a sibling, pulling down the parent node between them.

Worst Case: $\mathcal{O}(\log n)$ or $\mathcal{O}(\log k)$ for n nodes and k keys.

RED-BLACK TREES

A Red-Black Tree is a BST with

1. Each node either red or black
2. The root is always black
3. If a node is red, its children must be black
4. `null` pointers at the bottom are treated as black nodes
5. Every path from `root` to `null` contains the same number of black nodes.

We barely talked about these.

AA TREES

Are isomorphic to 2-3 Trees. They follow all the same rules as above, but add one more.

6. A red node may only be a right child of a black node.

1. Height h increases from bottom to top of tree. All leaves point to `null` nodes which are at level 0.
2. Red nodes are at the same level as parent.

Restructuring

1. `skew`: Right Rotation. Fixes left red child by making it black. Its black parent now becomes its right red child.
2. `split`: Left Rotation. Promotes the right child up a level. The two children are always black.
3. `update_level`: Checks that a node is at most one level higher than its lowest child. If this isn't the case, it pulls it down by making all children red.

Insert

First insert as with BST, insisting that the node be red. Then, cleanup as follows

1. If red left child of node n: `skew`(n)
2. If red right child of a red node n: `split`(n)

You might need to do this several times.

Delete

Let n be the node to delete.

1. Replace n with its inorder successor n'.
   n' is necessarily on level 1 now.
2. Delete n'. If it had a red right child, promote it to where n' was. If not, see Fixing the Tree.

Fixing the Tree

For each node p on the path from n' to the root,

1. If p has distance more than 1 from *any* of its children: `update_level`(p).
2. If p, or $p.right$, or $p.right.right$ has a left right child: `skew` on p, or $p.right$, or $p.right.right$ respectively.
3. If p or $p.right$ has a chain of 2 right red children: `split` on p, or $p.right$ respectively.

Repeat the above on every node that gets pulled down on the path to the root.

TREAPS

Each node stores $\left(\frac{k}{p}\right)$ where k is the key and p is the random priority.
Acts like a BST on k, but a max heap on p. In other words, for all nodes n

1. All nodes in left subtree have lesser keys.
2. All nodes in right subtree have greater keys.
3. Both children have lower priority.

Insert

Given a key k

1. Assign it a random priority p
2. Insert it as with BST
3. Patch it up using rotations, retaining the max queue property

Worst Case: $\mathcal{O}(n)$, **Average Case**: $\mathcal{O}(\log n)$

Delete

1. Find the key k as with a BST
2. Assign it priority $-\infty$
3. Rotate it to the bottom
4. Chop it off

Worst Case: $\mathcal{O}(n)$, **Average Case**: $\mathcal{O}(\log n)$

SCAPEGOAT TREES

Parameters

1. α: Balance threshold. Closer to 1 means can look more like a linked list. By default, $\alpha = \frac{2}{3}$.
2. n: Number of nodes in the tree.
3. m: Number of inserts in the tree. $m \geq n$

A node is a scapegoat if $\frac{size(n.child)}{size(n)} > \alpha$

Restructuring

1. Get inorder traversal of the tree
2. Pick out the $\lfloor n/2 \rfloor$ element to be the root. This is the **upper middle** for 0 indexing.
3. Repeat 2 for the induced left and right sub-lists.

Search

Same as standard BST.
Worse Case: $\mathcal{O}(\log n)$

Insertion

Worse Case: $\mathcal{O}(n)$
Let x be the node to insert.

1. Insert x as with standard BST.
2. Increment m and n
3. If $depth(x) > \log_{1/\alpha}(m)$
   i. Go back up the tree from x until you find a scapegoat s
   ii. `restructure`(s)

Deletion

Worse Case: $\mathcal{O}(n)$

1. Delete as with standard BST.
2. Decrement n
3. If $m > 2n$
   i. `restructure`$(root)$
   ii. set $m = n$

SPLAY TREES

Operations

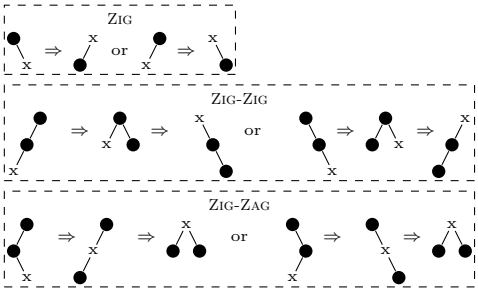
1. `zig`: Left or Right Rotation.
2. `zig-zig`: Left-Left or Right-Right Rotation. Rotate grandparent first, then parent.
3. `zig-zag`: Left-Right of Right-Left Rotation. Rotate parent first, then grandparent.

Splaying a node

Let x be the node to splay

- If x is a child of the root: `zig`
- If x is a left-left or right-right child: `zig-zig`
- If x is a left-right or right-left child: `zig-zag`

CMSC420 Cheatsheet



Keep doing this until the node is at the top.

Search

Let k be the target key we are searching for.

Call `splay`(k). The result is either

1. The key itself
2. Its inorder successor
3. Its inorder predecessor

Average Case: $\mathcal{O}(\log n)$

Worst Case: $\mathcal{O}(n)$

Insert

Let k be the key to insert.

Call `splay`(k), one of two things can happen

1. k is now at the root. If so, error
2. y is now at the root (the inorder pred/succ of k). Then let k be the new root and attach y as its child.

Average Case: $\mathcal{O}(\log n)$, **Worst Case**: $\mathcal{O}(n)$

Delete

Let k be the key to delete.

Call `splay`(k), one of two things can happen

1. k is not at the root. If so, error
2. k is now at the root with left subtree L and right subtree R.
   (a) If either R or L is empty, delete x and set the non-empty sub-tree as the root.
   (b) Otherwise, let T be the non-empty subtree.
      i. Call T.`splay`(k) to bring up inorder pred/succ y.
      ii. One of the subtrees of y will be empty. Delete k and set y to be the new root.

Average Case: $\mathcal{O}(\log n)$, **Worst Case**: $\mathcal{O}(n)$

SKIP LISTS

A multi-lane linked list.

Parameters

1. L: Maximum number of levels. Treated as constant
2. n: Number of nodes

Probabilities

Let $0 < p < 1$ be the probability of adding a level.

1. Probability that at least one node reaches level $L \geq 0$ or more: $(1 - (1 - p^L))^n$
2. Probability that at least one node reaches level $L \geq 0$ is bounded by np^L
3. Expected max level for n nodes: $\Theta(\log_{1/p} n)$
4. Expected max level for n nodes: $1 + \log_{1/p} n$
5. Expected number of levels: $2 + \log_{1/p} n$

Insert

Let x be the key to insert.

1. Go to x.
   i. Take the fastest lane (the highest) and stop at the largest key less than x. Record all pointers as you go
   ii. Go down by one lane and repeat step 1, until you are at the lowest lane.
2. Insert x.
   i. Flip a probability p coin until you get 0, or you've flipped it L times. Add as many lanes to x as the number of times you've flipped it.
   ii. Add the recorded pointers to x.
   iii. update the previous pointers to now point to x.

Average Case: $\mathcal{O}(\log n)$, **Worst Case**: $\mathcal{O}(n)$

Delete

1. Go to x, using the same technique as above, still recording pointers as you go.
2. Record all pointers of x.
3. Delete x.
4. Update all pointers pointing to x to now point to where x was pointing to.

Note: Fixing the number of pointers is $\mathcal{O}(1)$

Average Case: $\mathcal{O}(\log n)$, **Worst Case**: $\mathcal{O}(n)$

KD-TREES

Height

1. Best Case: $\mathcal{O}(\log n)$
2. Average Case: $\mathcal{O}(\sqrt{n})$
3. Worst Case: $\mathcal{O}(n)$

Search

1. Root splits on first dimension axis α, x for 2D or 3D.
2. In case of ties for axis α, go right.
3. Each layer splits by the next axis.
   i. 2D: $x \to y \to x \to \cdots$
   ii. 3D: $x \to y \to z \to x \to \cdots$

Worst Case: $\mathcal{O}(n)$ since we don't re-balance at all.

Insert

Let n be the node to insert.

1. Go through tree as if searching for n.
2. Insert as BST.

Worst Case: $\mathcal{O}(n)$ since we don't re-balance at all.

Delete

Let n be the node to delete.

1. Search for n in the tree.
2. If n is a leaf, delete it and you're done.
3. If n is not a leaf, say n splits by α. Replace n with the node n' with minimum α in the *right subtree* of n.
4. If n has no right subtree, it *must* have a left subtree. Replace n with the node n' in the left subtree with minimum α. Update the left subtree to become the right subtree.
   Note: n' might not be a leaf, n' might not split by α. That's fine.
5. Recursively call `delete`(n').

Worst Case: $\mathcal{O}(n)$ since we don't re-balance at all.

EXT. KD-TREES

Parameters

1. m: The maximum leaf size.

Split Method

1. **Cycle Split**: Split by $x \to y \to z \to x \to \cdots$
2. **Spread Split**: Split by axis α with largest spread.

Search

Let n be the node to search.

1. Follow splitting nodes down the tree.
2. If a splitting node has same axis value as n, check both subtrees.

Insert

Let n be the node to insert.

1. Follow splitting nodes down the tree.
2. If a splitting node has same axis value as n, **only go right**.
3. Insert into leaf.
   i. If leaf size is less than or equal to m, done.
   ii. If leaf size is more than m, split by Split Method.

Delete

Let n be the node to delete.

1. Follow splitting nodes down the tree as if searching for n.
2. If leaf becomes empty, delete splitting node and merge with sibling.

KNN

Let P be the coordinate, let k be the number of neighbors, let $L = []$ be the list of nearest neighbors, let $d^* = \max_{p \in L} \texttt{dist}(p,\ P)$.

1. If we are at a leaf, add every node to L, sort by distance from P, only keep first k.
2. If we are at a splitting node
   (a) If the list is not full, recurse to child with closest bounding box, if tied, prefer left. When checking both subtrees L then R, check `dist`(d^*, R) again before checking R, as d^* could have changed from L.
   (b) If the list is full, compute d^*.
      i. If either subtree's bounding box is closer to P than d^*, check it. In case of ties, check left first. When checking both subtrees L then R, check `dist`(d^*, R) again before checking R, as d^* could have changed from L.

TRIES

Only one character per branch.

Search

Follow the prefixes until you reach a \$ node.

Insert

Follow the tree until we fall out. Then build out a new branch of the tree if necessary.

Delete

Follow the path to the leaf, then travel back to the furthest ancestor which has more than one (non-null) child and remove the corresponding branch.

COMPRESSED TRIES

Insert

Let s be the string to insert.

1. If a prefix of s fully matches a branch, follow it and recurse with the rest of s.
2. If no prefix of s matches any branch, make a new branch.
3. If a prefix of s partially matches a branch b, split b by the largest shared component, add a branch with the rest of s and add a branch with the rest of b with its children.

Delete

Let s be the string to delete.

1. Remove the final branch corresponding to s.
2. If the parent of that branch only has one child, merge the two edges into one.

Height Bounds

Let h be the height of the trie.

1. $h \leq 1 +$ max word length. Good if you have *many*, *short* words.
2. $h \leq 1 + n$. Good if you have *few*, *long* words.

HASH FUNCTIONS

Parameters

1. h: A hash function.
2. n: Number of keys inserted.
3. m: Size of the hash table.
4. $\lambda = n/m$: Load factor of the hash table.

Addressing

1. **Closed Addressing**: Entries of the hash table point to a data structure.
2. **Open Addressing**: Keys are stored directly in the hash table.

Rebuilding

Rebuild after insert, if $\lambda > \lambda_{\max}$. The new size of the hash table is $m' = \lceil 2n/(\lambda_{\min} + \lambda_{\max}) \rceil$, where $0 \leq \lambda_{\min}, \lambda_{\max} \leq 1$.

We also rebuild after a delete takes us out of the acceptable range: $\lambda < \lambda_{\min}$. If we delete and we were already below acceptable range, we don't rebuild.

Probing

Only occurs with Open Addressing.

Method by which we find an open spot in the Hash Table if there is a collision. Linear Probing hits every slot, quadratic probing hits at least $\left\lfloor \frac{m}{2} \right\rfloor$ slots if m is prime.

BLOOM FILTERS

Parameters

1. B: The bit array.
2. k: Number of hash functions.
3. m: Size of the bloom filter.
4. n: Number of inserts.

Insert

Let s be the inputted value. For each hash function h_k, set $B[h_k(s)] = 1$.

Time Complexity: $\Theta(1)$, k is a constant.

Probabilities

Probability of false positive: $p = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k$.

Deletion

No deletion since it could introduce false negatives.

Rebuilding

Rebuild as with hash function.

DISJOINT SET DATA STRUCTURES

A forest where every node points to its parent. Roots point to themselves.

`find_rep`

For a node v, `find_rep`(v) is itself if v.`parent` = v. Otherwise, `find_rep`(v) = `find_rep`(v.`parent`).

If `find_rep`(v) = `find_rep`(u), then v and u are in the same tree.

Path Compression

For a node n, let r be the root pointer for the tree. When `find_rep`(n) is called, every node on the path back from n to r has its parent pointer updated to r.

Weighted Union

For two trees T_1 and T_2, with $|T_1| < |T_2|$, set `parent`(r_1) = r_2 where r_i is the root of T_i.

Time Complexity: $\mathcal{O}(\alpha(n))$ where $\alpha(n)$ is the Inverse Ackermann function, $\alpha(n) < 5$ for any n that can fit.