# Asymptotic Analysis

For functions $f(n)$, $g(n)$: $f(n) = O(g(n))$ if there exist $m, x_0$ such that

$$|f(n)| \leq m \cdot |g(n)| \text{ for all } x > x_0.$$

Similarly, $f(n) = \Omega(g(n))$ if there exist $m, x_0$ such that

$$|f(n)| \geq m \cdot |g(n)| \text{ for all } x > x_0.$$

$f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n)$ and $f(n) = \Omega(g(n))$

# Doc Dist

**Problem:** Find how similar two documents (list of strings) are.

**Approach:** Transform these documents into vectors where each dimension corresponds to a word and the magnitude of that dimension corresponds to the frequency of the word in that document. Once the two documents have been transformed into vectors, we can quantify the similarity of the documents by finding the angle between their corresponding vectors like so

$$\theta(D_1, D_2) = \arccos \frac{D_1 \cdot D_2}{||D_1|| \cdot ||D_2||}$$

The smaller the angle, the more similar two vectors are. During implementation in python, these are some of the optimizations we made:

- Using the list operation `extend` instead of list concatenation (+)
- Counting word frequencies using dictionary instead of lists of (word, frequency) tuples
- Using merge sort instead of insertion sort
- Using the dictionary to calculate dot product instead of creating sorted (word, frequency) tuples

# Peak Finding

## 1D Peak Finding Variant

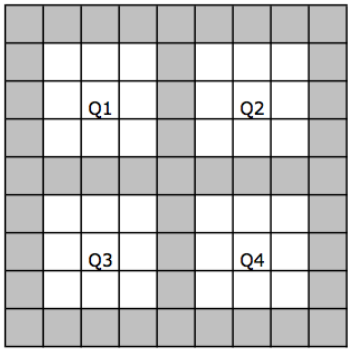
**Problem:** Given a list of $n$ numbers, find a peak element, defined as an element that is greater than or equal to its neighbors.

**Approach:** Look at the middle element and check to see if it's a peak. If so, we found a peak. Otherwise, it must have a neighboring element that is greater. Recurse into the half that contains this larger neighbor (this half must contain a peak) and repeat until a peak is found.

**Analysis:** Recurrence is $T(n) = T(n/2) + O(1)$. Solve to get a total runtime of $O(\log n)$

## 2D Peak Finding Variant

**Problem:** Given a $n \times n$ matrix of numbers, find a peak element, defined as an element that is greater than or equal to its neighbors.




"Window frame" of a matrix forming four quadrants

**Approach:** Take the maximum element found in the matrix "window frame" defined as columns 0, $\frac{n}{2}$, and $n-1$ and rows 0, $\frac{n}{2}$, and $n-1$. Check to see if this maximum element is a peak. If so, we found a peak. Otherwise, it must have a neighboring element that is greater. Recurse into the quadrant that contains this larger neighbor (this quadrant must contain a peak) and repeat until a peak is found.

**Analysis:** Recurrence is $T(n) = T(n/4) + O(n)$. Solve to get a total runtime of $O(n)$

# Binary Search Trees

## BST Structure

Binary search trees are made up of nodes that contain the following parameters:

- $x$.key - Value stored in node $x$
- $x$.left- Pointer to the left child of node $x$. NIL if $x$ has no left child
- $x$.right - Pointer to the right child of node $x$. NIL if $x$ has no right child
- $x$.parent - Pointer to the parent node of node $x$. NIL if $x$ has no parent, i.e. $x$ is the root of the tree

**BST property:** Each node has the property that every key found in the node's left subtree is less than or equal to the key at that node and every key found in the node's right subtree is greater than or equal to the key at that node.

## BST Operations

Binary search trees have the following operations that allows users to search for elements and manipulate the tree.

`search(k)`

**Description:** Find key $k$. Return the node that contains $k$ if it exists or NIL if it is not in the tree.

**Approach:** Starting at the root, check to see if the node we're at contains key $k$. If so, return that node. If not, traverse to the left child if $k$ is smaller or traverse to the right child if $k$ is larger and repeat. If we reach NIL before finding $k$, then $k$ is not in the tree and we return NIL.

**Analysis:** Searching could potentially go along the longest branch of the tree, so the runtime of `search` is $O(h)$ where $h$ is the height of the tree.

`insert(k)`

**Description:** Insert key $k$ into the tree.

**Approach:** Starting at the root, compare the key of the node we're at to $k$. Traverse to the left child if $k$ is smaller or traverse to the right child otherwise and repeat until we reach NIL. Once we reach NIL, replace the NIL node with a node containing $k$, thus inserting $k$ into the tree.

**Analysis:** Searching where to insert $k$ could potentially go along the longest branch of the tree, so the runtime of `insert` is $O(h)$ where $h$ is the height of the tree.

`find-min(x)`

**Description:** Find the node with the minimum key of the tree rooted at node $x$ and return it.

**Approach:** Starting at $x$, keep traversing to the left child until we reach a node with no left child. This node contains the minimum key, so we return it.

**Analysis:** The left-most branch could potentially be the longest branch of the tree, so the runtime of `find-min` is $O(h)$ where $h$ is the height of the tree.

`next-larger(x)`

**Description:** Find the node that contains the next larger key relative to the key at node $x$.

**Approach:** If $x$ has a right child, return find-min($x$.right). Otherwise, traverse upwards through the tree in $x$'s ancestry line until we reach a node with a larger key than $x$'s key and return this node.

**Analysis:** We could potentially go through the longest branch of the tree up $x$'s ancestry line or down $x$'s right subtree so the runtime of `find-min` is $O(h)$ where $h$ is the height of the tree.

`delete(x)`

**Description:** Remove node $x$ from the tree while maintaining the tree's properties

**Approach:** If $x$ has no children, just remove it by replacing $x$ with NIL. If $x$ has one child, splice out $x$ by linking $x$'s parent to $x$'s child. If $x$ has two children, splice out $x$'s successor and replace $x$ with $x$'s successor.

**Analysis:** Finding a successor using `next-larger` takes $O(h)$, so `delete` takes $O(h)$ as well.

`in-order-walk(`$x$`)`

**Description:** Traverses through the tree rooted at node $x$ in increasing order and prints out every key

**Approach:** Recursively call `in-order-walk` on $x$'s left child, then print out $x$'s key, and finally recursively call `in-order-walk` on $x$'s right child.

**Analysis:** Every node is visited once. `in-order-walk` takes $O(n)$ time where $n$ is the number of nodes in the tree rooted at $x$

# AVL Trees

AVL trees are balanced binary search trees where each node is augmented to have an additional height parameter, defined as the length of the longest branch of the tree rooted at that node. The height of a leaf node is 0 and the height of a NIL node can be treated as -1.

**AVL Tree property:** An AVL tree has the property that the height of the children of each node differ by at most one.

This property allows us to constrain the height of the tree to $\Theta(\log n)$, which is important since most of our operations had a runtime of $O(h)$. To prove this, let $n_h$ be the minimum number of nodes of an AVL tree of height $h$. To construct this minimum AVL tree, the root node must have a child of height $h-1$ and a child of height $h-2$. The total number of nodes in this minimum AVL tree is thus the sum of the sizes of the two children plus the root node itself. We get the following inequality:

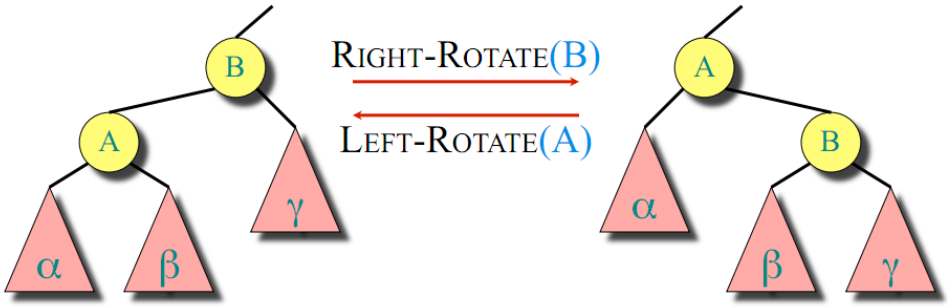
$$n_h \geq 1 + n_{h-1} + n_{h-2} \tag{1}$$
$$n_h > 2n_{h-2} \tag{2}$$
$$n_h > 2(2n_{h-4}) \tag{3}$$
$$n_h > 2(2(2(2...(2n_{0 \text{ or } 1})...) \tag{4}$$
$$n_h > 2^{h/2} \tag{5}$$
$$h < 2 \lg n_h \tag{6}$$

Showing that AVL tree's height is $O(\log n)$. The lower bound can be calculated similarly to show that height is $\Theta(\log n)$.

## AVL Rotations

Searching through an AVL tree is no different than searching through a binary search tree. However, when we manipulate the AVL tree through insertions and deletions, we have to use rotation operations to ensure that the AVL tree property is maintained.

Rotations move around nodes in an AVL tree, changing the heights of some nodes while maintaining the binary search tree property that preserves the ordering of the node's values. There are two rotations which are the reverse of each other:




Note that when a rotation is made, the heights of $A$ and $B$ must be updated to maintain correctness.

Insertion and deletion in an AVL tree begins exactly the same as an insertion and deletion in a binary search tree. Once a node is inserted or deleted though, you must check the node's ancestry line and make sure their height is correct, as the insertion or deletion may affect those heights. Also, an insertion or deletion may lead to a violation of the AVL tree property and cause the heights of a node's children to differ by two. When this happens, the proper rotations must be executed to restore the AVL tree property. Note that in the case of deletions, multiple rotations may be required since one rotation may cause an imbalance elsewhere in the tree.

# Tree Augmentation

An AVL tree's nodes are augmented to include a height parameter. Another augmentation example is subtree size. Every node contains a parameter `size` that represents the size of the subtree rooted at that node. Insertion in this augmented tree maintains the correctness of the `size` parameter by incrementing `size` of each node traversed as we look for where to insert the new key. Deletion in this augmented tree must decrement `size` of the ancestry line of the removed node.