# 1 Complexity / Asymptotic Analysis

Let $n$ denote the problem size, and $r(n)$ denote the amount of resources (time / space) needed to solve the problem of size $n$

**Note:** read $\forall\ n > n_0$ as "for some large value of $n$"

## 1.1 $\Omega$ "Big Omega" Notation

Provides a **lower bound** of $r(n)$

$r(n)$ has an order of growth $\Omega(g(n))$ if there is a positive constant $k$, and a number $n_0$ such that $k \cdot g(n) \leq r(n)\ \forall\ n > n_0$

## 1.2 "Big O" Notation

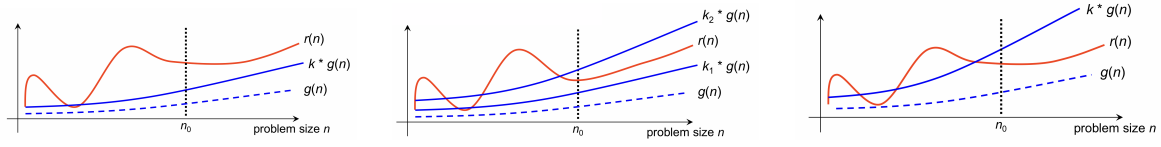
Provides an **upper bound** of $r(n)$

$r(n)$ has an order of growth $O(g(n))$ if there is a positive constant $k$, and a number $n_0$ such that $r(n) \leq k \cdot g(n)\ \forall\ n > n_0$

## 1.3 $\theta$ "Big Theta" Notation

Provides an **upper and lower bound** of $r(n)$

$r(n)$ has an order of growth $\theta(g(n))$ if there are positive constants $k_1$ and $k_2$, and a number $n_0$ such that $k_1 \cdot g(n) \leq r(n) \leq k_2 \cdot g(n)\ \forall\ n > n_0$




## 1.4 Recurrence Relations

Let $T(n)$ be the time complexity of the given function for the problem of size $n$. Then the recurrence relation for the function may be thought of as follows:

$$T(n) = T(operations\ done\ now)\ + T(deferred\ operations)$$

In general, $T(n) = \theta(n^k) + T(n-1) \rightarrow T(n) = \theta(n^{k+1})$

| Recurrence Relation | Time Complexity | Examples |
|---|---|---|
| $T(n) = \theta(1) + T(n-1)$ | $\theta(n)$ | build_list(f, n), append, linear search |
| $T(n) = \theta(n) + T(n-1)$ | $\theta(n^2)$ | selection sort, insertion sort |
| $T(n) = \theta(1) + 2T(\frac{n}{2})$ | $\theta(\log n)$ | Binary search, tree traversal |
| $T(n) = \theta(n) + 2T(\frac{n}{2})$ | $\theta(n \log n)$ | Merge sort |
| $T(n) = \theta(1) + 2T(n-1)$ | $\theta(2^n)$ | Tree recursive fibonacci |

**Master Theorem**

If T(n)= $aT(\frac{n}{b}) + f(n)$, $f(n) = n^k$, then

$$T(n) = \begin{cases} n^k, & a < b^k \\ n^k \log_b n, & a = b^k \\ n^{\log_b a}, & a > b^k \end{cases}$$

## 1.5 Calculating Orders of Growth

- Only the **largest** order of growth matters! i.e. $O(2^n) + O(n^{100}) = O(2^n)$
- If the problem size depends more than one variable, then their individual orders of growth **cannot be combined**! i.e. $O(m) + O(n^2) \neq O(n^2)$
- If $T(n)$ is a polynomial of degree $k$ then $T(n) = O(n^k)$
- If $T(n) = O(f(n))$ and $S(n) = O(g(n))$, then
  - $T(n) + S(n) = O(f(n) + g(n))$
  - $T(n) \cdot S(n) = O(f(n) \cdot g(n))$
- Loops and Nested Loops: $T(n) = (\#\ iterations) \cdot (max\ cost\ of\ single\ iteration)$
- Sequential Statements: $T(n) = \sum(cost\ of\ statements)$
- Conditionals: $T(n) = max(cost\ of\ conditional\ branches) \leq \sum(cost\ of\ branches)$

| Function | Name |
|---|---|
| 5 O(1) | Constant |
| loglog(n) | double log |
| log(n) | logarithmic |
| $\log^2(n)$ | Polylogarithmic |
| n | linear |
| nlog(n) | log-linear |
| $n^3$ | polynomial |
| $n^3\log(n)$ | |
| $n^4$ | polynomial |
| $2^n$ | exponential |
| $2^{2n}$ | |
| n! | factorial |

## 1.6 Useful Mathematical Equalities

### 1.6.1 Arithmetic Series

If $a_n = a_{n-1} + c$,

$$\sum_{i=1}^{n} a_i = a_1 + a_2 + a_3 + \cdots + a_n = \frac{n(a_n + a_1)}{2}$$

### 1.6.2 Geometric Series

$$\sum_{i=1}^{n} ar^{i-1} = a + ar + ar^2 + \cdots + ar^{n-1} = a \cdot \frac{1 - r^n}{1 - r}$$

If $0 < r < 1$,

$$\sum_{i=1}^{\infty} ar^{i-1} = \frac{a_1}{1 - r}$$

### 1.6.3 Harmonic Series

$$\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n} \approx ln(n+1)$$

### 1.6.4 Sum of Squares

$$\sum_{i=1}^{n} i^2 = 1 + 4 + 9 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

### 1.6.5 Logarithms

$log_b a = \frac{1}{log_a b}$

$log_a x = \frac{log_b x}{log_b a}$

$log_2(n!) = n \cdot log_2 n$

## 1.7 Examples

$O(7.2 + 34n^3 + 3254n) = O(n^3)$

$$\begin{aligned} O(n^2 logn + 25n log^2 n) &= O(nlogn(n + 25logn)) \\ &= O(nlogn) \cdot O(n + 25logn) \\ &= O(nlogn) \cdot O(n) \\ &= O(n^2 logn) \end{aligned}$$

$$\begin{aligned} O(2^{4logn} + 5n^3) &= O((2^{logn})^4) + O(n^3) \\ &= O(n^4) + O(n^3) \\ &= O(n^4) \end{aligned}$$

$$\begin{aligned} O(2^{2n^2+4n+7}) &= O(2^{2n^2}) \cdot O(2^{4n}) \cdot O(2^7) \\ &= O((2^{n^2})^2) \cdot O((2^n)^4) \cdot O(1) \\ &= O(2^{2n^2+4n}) \end{aligned}$$

$$\begin{aligned} &O(\frac{n^2}{17} \cdot \frac{\sqrt{n}}{4} + \frac{n^3}{n-7} + n^2 \cdot logn) \\ &= O(n^2 \cdot \sqrt{n} + \frac{n^3}{n-7} + n^2 \cdot logn) \\ &= O(n^{2.5} + n^2 + n^2 \cdot logn) \\ &= O(n^{2.5}) \end{aligned}$$

$T(n)$ is the running time of a divide-and-conquer algorithm that divides the input of size $n$ into $n/10$ equal-sized parts and recurses on all of them. It uses $O(n)$ work in dividing/recombining all the parts (and there is no other cost). The base case for the recursion is when the input is less than size 20, which costs $O(1)$.

$\#\ parts = \frac{n}{10}$
$size\ of\ 1\ part = \frac{n}{n/10}$

$$\begin{aligned} T(n) &= \frac{n}{10} \cdot T(\frac{n}{n/10}) + O(n) \\ &= \frac{n}{10} \cdot T(10) + O(n) \\ &= \frac{n}{10} \cdot O(1) + O(n) \\ &= O(n) \end{aligned}$$

# 2 Binary Search

```
def search(A, key, begin, end):
    if (begin > end): return -1
    # avoid integer overflow errors
    mid = begin + (end-begin)/2
    if (key < A[mid]):
        # eliminate right half
        return search(A, key, begin, mid)
    else if (key > A[mid]):
        # eliminate left half
        return search(A, key, mid+1, end)
    else: return mid
```

## 2.1 Algorithm Design

### 2.1.1 Functionality

- What the algorithm is supposed to do

Binary search functionality:

1. If `key` is in `A` ,return the index of `key`
2. Else, return -1

### 2.1.2 Precondition

- Fact that is true when the function begins
- Usually important for it to work correctly.

Binary search preconditions:

1. `A` is of size $n$
2. `A` is sorted

### 2.1.3 Postcondition

- Fact that is true when the function ends
- Useful to show that the computation was done correctly

Binary search postconditions:

1. If `key` is in `A`: `A[begin] = key`

### 2.1.4 Invariant

- Relationship between variables that is always true.
- Loop invariant: relationship between variables that is true at the start (or end) of each iteration

Binary search invariant:

1. `A[begin] < key < A[end]`
2. i.e. `key` is always in the search range

## 2.2 Applications

Binary search may be used for other problems besides searching arrays, consider the following problem:

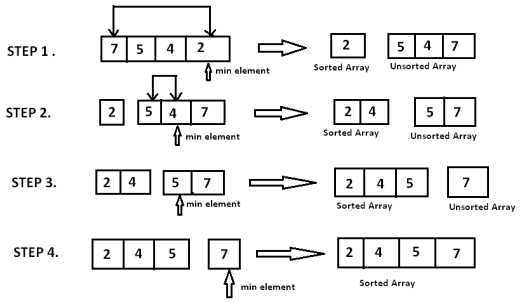
1. Given a function `complicatedFunction(input)` that is **monotonic increasing**
2. i.e. `complicatedFunction(i) < complicatedFunction(i+1)`
3. Task: Find the minimum value `j` such that: `complicatedFunction(j) > <some number>`

**Note:** The opposite is also applicable (e.g. finding max given **monotonic decreasing** function)

# 3 Sorting

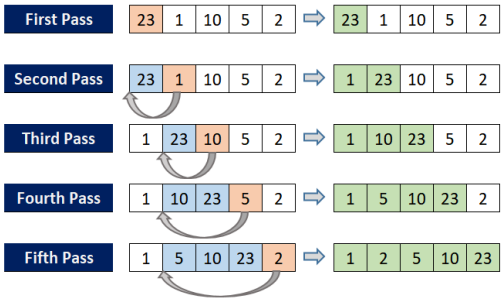
## 3.1 Selection Sort

```
def selectionSort(A):
    for j in [1:len(A)]:
        k = indexOfMinElement(A[j..len(A)])
        swap(A[j], A[k])
```



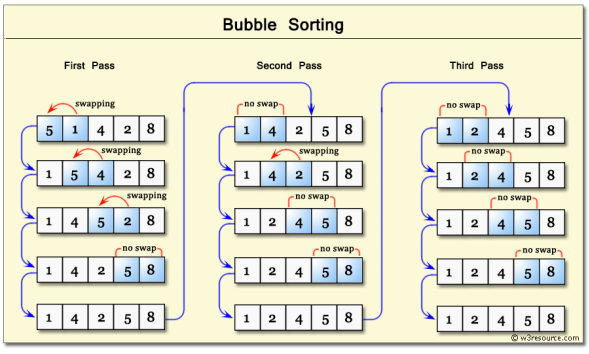

## 3.2 Insertion Sort

```
def insertionSort(A):
    for i in [1:len(A)]:
        key = A[i]
        j = i - 1
        # find correct position for key wi
        while (j >= 0) and (A[j] > key):
                # move element to the righ
            A[j+1] = A[j]
            j -= 1
        # insert key here
        A[j+1] = key
```



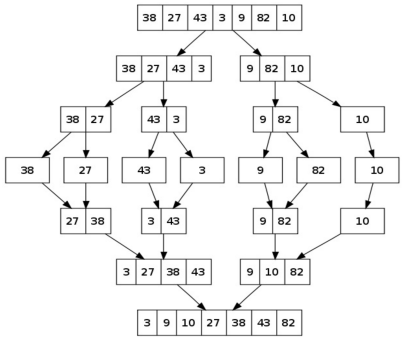

## 3.3 Bubble Sort

```
def bubbleSort(A):
    repeat (until no swaps):
        for j in [0 : len(A)-1]:
            if (A[j] > A[j+1]):
                swap(A[j], A[j+1])
```



## 3.4 Merge Sort

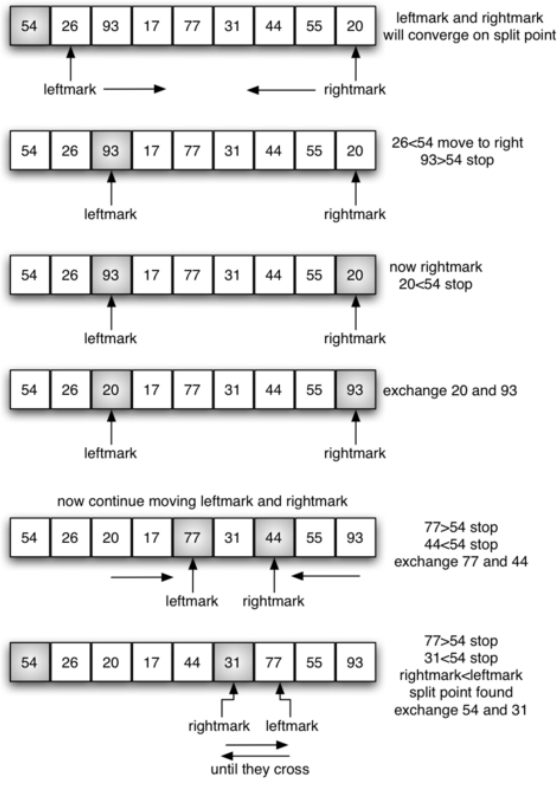
```
def mergeSort(A):
    if (len(A) <= 1):
        return
    else:
        mid = len(A) // 2
        left = mergeSort(A[0:mid])
        right = mergeSort(A[mid:len(A)])
        return merge(left, right)
```




## 3.5 Quick Sort

```
def quickSort(A, n):
    if (n <= 1): return
    else:
        Choose pivot index p
        newP = partition(A[1:n], n, p)
        x = quickSort(A[1:newP], newP-1)
        y = quickSort(A[newP+1:n], n-newP)


def partition(A, n, p)
    pivot = A[p]
    swap(A[1], A[p])
    low = 2
    high = n + 1
    while (low < high)
        while (A[low] < pivot) and (low < high):
            low += 1
        while (A[high] > pivot) and (low < high):
            high -= 1
        if (low < high):
            swap(A[low], A[high])
    swap(A[1], A[low-1])
    return low - 1
```



## 3.6 Properties

| ***Algorithm*** | ***Stability*** | ***In-place*** | ***Invariant*** |
|---|---|---|---|
| Selection | ✗ | ✓ | At the end of iteration j: the j smallest items in the array are sorted. |
| Insertion | ✓ | ✓ | At the end of iteration j: the first j items in the array are in sorted order. |
| Bubble | ✓ | ✓ | At the end of iteration j: the j largest items in the array are sorted. |
| Merge | ✓ | ✗ | N.A |
| Quick | ✗ | ✓ | 1. Pivot is in correct position at the end of partitioning.<br>2. For all `1 < i < low, A[i] < pivot`<br>3. For all `j >= high, A[j] > pivot` |

## 3.7 Time Complexity

| ***Algorithm*** | ***Unsorted*** | ***Sorted*** | ***Reverse Sorted*** | ***Almost Sorted*** |
|---|---|---|---|---|
| Selection | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| Insertion | $O(n^2)$ | $O(n)$ | $O(n^2)$ | $O(n)$ |
| Bubble | $O(n^2)$ | $O(n)$ | $O(n^2)$ | $O(n^2)$ |
| Merge | $O(nlogn)$ | $O(nlogn)$ | $O(nlogn)$ | $O(nlogn)$ |
| Quick | $O(nlogn)$ | $O(n^2)$ | $O(n^2)$ | $O(nlogn)$ |

**Swaps**

| ***Algorithm*** | ***Best Case*** | ***Worst Case*** |
|---|---|---|
| Selection | 0 (Sorted) | $O(n)$ |
| Insertion | 0 (Sorted) | $O(n^2)$ (Reverse) |
| Bubble | 0 (Sorted) | $O(n^2)$ (Reverse) |
| Merge | 0 | 0 (Only Copying) |
| Quick | $O(nlogn)$ | $O(n^2)$ (Reverse) |

**Comparisons**

| ***Algorithm*** | ***Best Case*** | ***Worst Case*** |
|---|---|---|
| Selection | $O(n^2)$ | $O(n^2)$ |
| Insertion | $0(n^2)$ (Sorted) | $O(n^2)$ |
| Bubble | $0(n^2)$ (No Flag) | $O(n^2)$ |
| Merge | $O(nlogn)$ | $O(nlogn)$ |
| Quick | $O(nlogn)$ | $O(n^2)$ |

## 3.8 Space Complexity

| ***Algorithm*** | ***Extra Memory*** |
|---|---|
| Selection | $O(1)$ |
| Insertion | $O(1)$ |
| Bubble | $O(1)$ |
| Merge | $O(nlogn)$ |
| Quick | $O(n)$ (average: $O(logn)$) |

# 4 Binary Search Trees

A BST is made up of nodes or vertices. For each node, define:

- `node.left = left child of node`
- `node.right = right child of node`
- `node.parent = parent of node`
- `node.key = value of node`

The BST property is as follows:

For any two nodes `u` and `v`,

- `u.key < v.key if u is in the left subtree of x`
- `u.key >= v.key if u is in the right subtree of x`

## 4.1 Operations

### 4.1.1 Search

```
def search(key, root):
    if (root == null): return -1
    else if (key > root.key):
        # eliminate left half
        return search(key, root.right)
    else if (key < root.key):
        # eliminate right half
        return search(key, root.left)
    else:
        return root
```

Time Complexity: $O(tree\ height)$

### 4.1.2 Insert

We use the same logic as in search, except that we only stop when we find a null node. Once found, we create a new node and insert it.
Time Complexity: $O(tree\ height)$

### 4.1.3 Find Max & Min

For max, recursively visit the right child until we reach a node with no right child, then return that node's value.

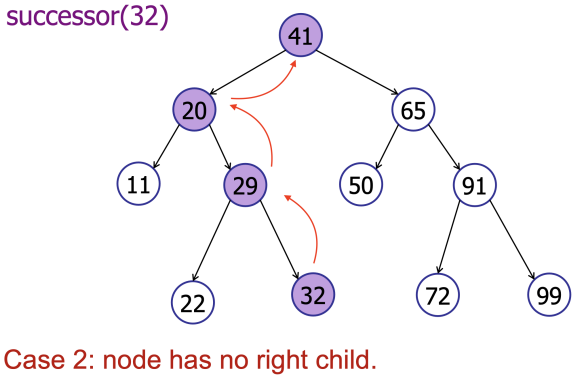
For min, recursively visit the left child until we reach a node with no left child, then return that node's value.

Time Complexity: $O(tree\ height)$

### 4.1.4 Successor & Predecessor

Successor / Predecessor to a node `u` is the immediate node with a value higher / lower than `u.key`

```
successor(u):
    if (u.right != null):
        return findMin(u.right)
    else:
        p = u.parent
        while (p != null && u == p.right):
            u = p
            p = u.parent
        if (p == null): return -1
    else: return p
```



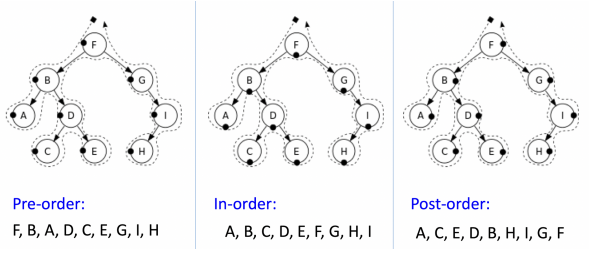

- If `u` has a right child, then the min of its right subtree would be the successor
- Else, we try to traverse up the tree to find an ancestor of `u` that is a left child
  - Why left child? So long as the ancestor is a right child, we get smaller values as we go upwards.
  - However, the moment there is a "right turn", we have found an ancestor whose value is larger than everything before it.
  - That ancestor would then be the successor to `u`, as `u` would be its predecessor i.e. max value of its left subtree.
- Time Complexity: $O(tree\ height)$

**Note:** The above is symmetric for `predecessor(u)`

### 4.1.5 Traversals

- Preorder
  1. Display data of the root
  2. Recursively traverse the left subtree
  3. Recursively traverse the right subtree
- Inorder
  1. Recursively traverse the left subtree
  2. Display data of the root
  3. Recursively traverse the right subtree
- Postorder
  1. Recursively traverse the left subtree
  2. Recursively traverse the right subtree
  3. Display data of the root

Notice the use of the dots drawn on each vertex



### 4.1.6 Delete

To delete a node with children effectively, it is best to replace it with another node such that minimal bubbling up is required. In the case of deletion, the successor of the node being deleted is the best candidate to use for replacement.

There are 3 cases when it comes to deletion of a node `u`:

1. `u` has no children
    - Remove `u` – O(1)
2. `u` has 1 child
    - Connect `u.child` to `u.parent` – O(1)
    - Remove `u` – O(1)
3. `u` has 2 children
    - `v = successor(u)` – $O(tree\ height)$
    - Replace `u.key` with `v.key` – O(1)
    - Recursively delete `v` in `u.right` – $O(tree\ height)$
    - Overall - $O(tree\ height)$
    - But why successor? Because the successor has at most 1 child, which makes it easier to delete.
        - If `u` has 2 children, then `u` must have a right child
        - `successor(u)` must be the minimum of the `u.right`
        - A minimum element of a BST has no left child
        - $\therefore$ `successor(u)` has at most 1 child

### 4.1.7 Rank

The rank of an element is its position relative to the sorted order, i.e. the kth smallest item would have a rank of k. To calculate that, we need to start from the root:

```
// computes the rank of `u` within the subtree rooted at `root`
getRank(u, root):
    if (u.key < root.key):
        return getRank(u, root.left)
    else if (u.key > root.key):
        return root.left.weight + 1 + getRank(u, root.right)
    else:
        return root.left.weight + 1
```

#### 4.1.8 Select

Select is the reverse of rank. Given a rank, return the value of the node with that rank. We will need to start checking from the root:

```
select(rank, root):
    // this is equivalent to calling getRank(root, root)
    rankOfRootInSubTree = root.left.weight + 1
    if (rank < rankOfRootInSubTree):
        return select(rank, root.left)
    else if (rank > rankOfRootInSubTree):
        // eliminate the root and its right subtree
        return select(rank - rankOfRootInSubTree, root.right)
    else:
        return root
```

Time Complexity: $O(tree\ height)$

## 4.2 AVL (Balanced) Trees

An AVL Tree's principle is to keep itself **height-balanced** via rotations, such that all operations that depend on $O(tree\ height)$ results in $O(logn)$.
To do so, we need to maintain two more (recursively defined) attributes with each node:

1. Weight: Number of nodes in the subtree rooted at the node.
   (a) `weight(null) = 0`
   (b) `weight(leaf) = 1`
   (c) `weight(u) = w.left.weight + w.right.weight + 1`
2. Number of edges on the path from the node to the deepest leaf.
   (a) `height(empty tree) = -1`
   (b) `height(u) = max(u.left.height, u.right.height) + 1)`

### 4.2.1 Balanced & Height-Balanced

- BST is balanced if $h = O(logn)$, i.e. $c \cdot logn$, allowing all operations to run in $O(logn)$ time
- A node `u` is said to be height-balanced if `|u.left.height - u.right.height|` $\leq 1$.
- BST is height-balanced if every node is height-balanced

**Note:** Height-balanced $\rightarrow$ balanced, balanced $\not\rightarrow$ height-balanced
**Note:** A height-balanced tree has height $O(2logn) = O(logn)$

### 4.2.2 Maintaining Height-Balanced Property

Define the balance factor of a node `u` as `balance(u) = u.left.height - u.right.height` When `|balance(u)|` $\geq 2$, rebalancing is required. This must be done from the insertion/deletion point **up to the root**.

### 4.2.3 Tree Rotations

When rotating, operations must be done in the correct order to prevent accidental dereferencing.

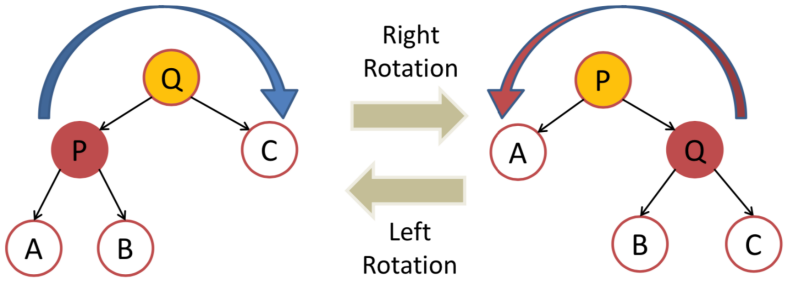
#### Right Rotation

```
rightRotate(u):
    // Create a pointer to q.left
    p = q.left
    p.parent = q.parent
    q.parent = p
    // At this point, we have swapped their parents.
    // Now, we need to swap their children.
    q.left = p.right // passing of b from p to q
    if (p.right != null):
        p.right.parent = Q
    p.right = q
    // Make sure to update height of q then p.
    return p
```

**Left Rotation**

```
leftRotate(u):
    // Create a pointer to p.right
    q.parent = p.parent
    p.parent = q
    // At this point, we have swapped their parents.
    p.right = q.left // passing of b from q to p
    if (q.left != null):
        q.left.parent = p
    q.left = p
    // Make sure to update height of q then p.
    return q
```

**Note:** right rotations require a left child, left rotations require a right child




### 4.2.4 Rebalancing

A node is **left-heavy** if its left subtree is **taller** than the right sub-tree, i.e. `node.left.height > node.right.height`.

1. If `v` is out of balance and left heavy:

   `v.left` is **balanced**: `rightRotate(v)`

   `v.left` is **left-heavy**: `rightRotate(v)`

   `v.left` is **right-heavy**: `leftRotate(v.left)` **and** `rightRotate(v)`

2. If v is out of balance and right heavy:

   Symmetrical cases, e.g. if `v.right` is **balanced**: `rightRotate(v)`

- The maximum number of rotations required upon insertion is 2, while for deletion it is $O(logn)$

# 5 Augmented AVL Trees

In addition to storing keys, we may augment an AVL tree to support various search functions.

## 5.1 Maximum Value

**Augmentation:** Each node `u` is augmented with: `value`, that specifies the value associated with that node, and `max`, the maximum `value` for any node in the subtree rooted at `u`

**New Operations**

`updateValue(key, newValue)`

1. Search the tree in the usual way for the specified `key`
2. Assuming a node `u` was found, update `u.value = newValue`
3. Update the tree - for every node `v` on the path from `u` to `root`, update `v.max = max(v.left.max, v.right.max, v.value)`
   **Note:** Make sure to check children for `null`

`getMax()`

1. Let `v = root`, `max = root.max`
2. If `v.value == max`, return `v`
3. If `v.left.max == max`, recurse on `v.left`
4. Else recurse on `v.right`
   **Note:** The order of our checks is significant as they determine the tiebreaker by key

**Maintenance**

- When performing a rotation on `u`, only `u` and `u.parent` change. Let `v = u.parent`. After a rotation of `u`, set `u.max = v.max`, and `v.max = max(v.points, v.left.points, v.right.points)` (avoiding `NullPointerException`)
- When a node `u` is inserted: Set `u.value = <initial>` and `u.max = <initial>`. Perform rotations to rebalance.
- When a node `u` is deleted:
  1. if `u` is a leaf, we can just delete it. For every ancestor `v` of `u`, update `v.max = max(v.left.max, v.right.max, v.points)`
  2. if `u` has one child, then delete `u`, connecting `u.parent` to `u.child`. For every node `v` on the path from `u` to `root`: `v.max = max(v.left.max, v.right.max, v.points)`
  3. node u has two children. Let `v = successor(u)`. Delete `v` from the tree, and for every node `w` on the path from `v` to `u`, update `w.max = max(w.left.max, w.right.max, w.points)`. Then replace `u` with `v`, and continue to update every node `w` on the path from `v` to the `root`.

  Perform rotations to rebalance.

## 5.2 Total Count

Assume that there is some `value` that needs to be stored, and also there is a function `isSpecial(value)` that is used to determine if a node is special.

**Augmentation:** Each node `u` is augmented with: `value`, that specifies the value associated with that node, and `total`, the count of the number of special nodes in the subtree rooted at `u`

**New Operations**
`addToValue(key, val)`

1. Search the tree in the usual way for the specified `key`
2. Assuming a node `u` was found, update `u.value += val`
3. If `isSpecial(u.value)`, update the tree - for every node `v` on the path from `u` to `root`, update `v.total += 1`

**Note:** The above is symmetrical for `subtractFromValue`

`searchSpecial()`

1. Let `v = root`
2. Base Case: if `isSpecial(v.value)`, return `v`
3. If `v.total == 0`, return `null`
4. Else if `v.isLeaf()`, return `v`
5. Else if `v.left.total > 0`, recurse on `v.left`
6. Else recurse on `v.right`

**Note:** Avoid `NullPointerException`

**Maintenance**
Similar to **Max Value** augmentation, except that a node `u` is updated as follows:
`u.total = u.left.total + u.right.total`