Orders of Growth

Recurrence Relation	Time Complexity	Examples
$T(n) = T(n-1) + \theta(1)$	$\theta(n)$	linear search
$T(n) = T(n-1) + \theta(n)$	$\theta(n^2)$	selection/insertion/bubble sort
$T(n) = T(\frac{n}{2}) + \theta(1)$	$\theta(\log n)$	Binary search
$T(n) = T(\frac{n}{2}) + \theta(n)$	$\theta(n)$	Quickselect
$T(n) = 2T(\frac{n}{2}) + \theta(1)$	$\theta(n)$	tree traversal
$T(n) = 2T(\frac{n}{2}) + \theta(n)$	$\theta(n \log n)$	Merge sort
$T(n) = 2T(n-1) + \theta(1)$	$\theta(2^n)$	
$T(n) = T(n-1) + T(n-2) + \theta(1)$	$\theta(\phi^n) \approx \theta(1.62^n)$	Tree recursive fibonacci
$T(n) = T(\sqrt{n}) + \theta(1)$	$\theta(log(logn))$	N.A
$T(n) = \sqrt{n}T(\sqrt{n}) + \theta(\sqrt{n})$	$\theta(nlog(logn))$	Let $n = 2^k$ and Master's Theorem

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

Master Theorem

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

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

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

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

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

If $0 < r < 1$

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

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

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

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

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

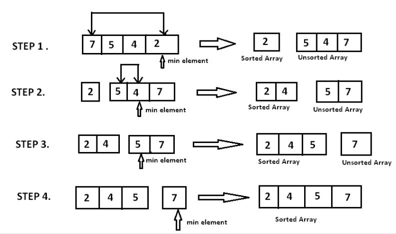
$$log_2(n!) = n \cdot log_2 n$$

Function	Name
1	constant
$log(logn)$	double log
$logn$	log
$log^2 n$	polylog
n	linear
$nlogn$	log-linear
$nlog^2 n$	
n^2	polynomial
$n^2 logn$	
2^n	exponential
3^n	
2^{2n}	
$n!$	factorial
$(n+1)!$	

Sorting

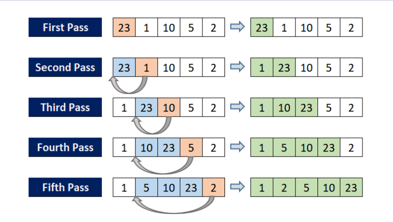
Selection Sort

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



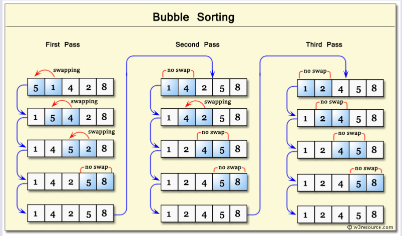
Insertion Sort

```
for i in [1:len(A)]:
    key = A[i]
    j = i - 1
    # find correct position for key within
    A[1:j]
    while (j >= 0) and (A[j] > key):
        # move element to the right ('make
        space for key')
        A[j+1] = A[j]
        j -= 1
    # insert key here
    A[j+1] = key
```



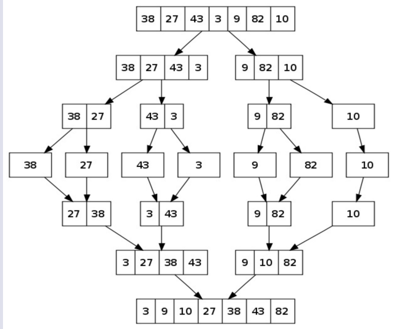
Bubble Sort

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

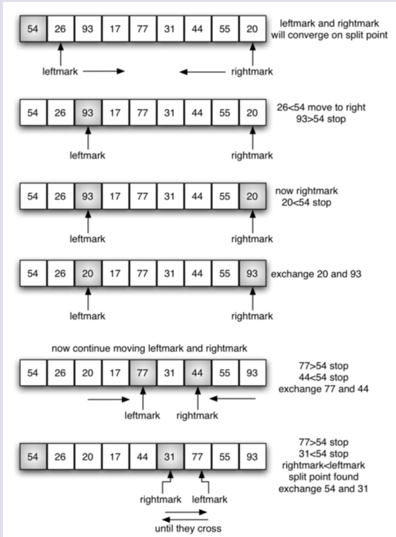


Merge Sort

```
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)
```



QuickSort



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**. The remaining elements are in their original order
Bubble	✓	✓	At the end of iteration j: **the j largest items in the array are sorted**.
Merge	✓	✗	If **elements from different halves have been swapped**, then the 2 halves have been *mergeSorted* & are in sorted order
Quick	✗	✓	1. **Pivot is in correct position at the end of partitioning.** 2. For all `1 < i < low, A[i] < pivot` 3. For all `j >= high, A[j] > pivot`

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	$O(nlogn)$	$O(n^2)$

Space Complexity

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

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
```

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) > num`

```
def bisectRight(A, key):
    # returns the index of the first value
    strictly greater than key
    low = 0, high = len(A) - 1
    if (A[high] < key): return -1
    while (low < high)
        mid = (low + high)/2
        if (A[mid] <= key): low = mid+1
        else if (A[mid] > key): high = mid
    return low
```

AVL Trees

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)`

- 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|` ≤ 1.
- BST is height-balanced if every node is height-balanced

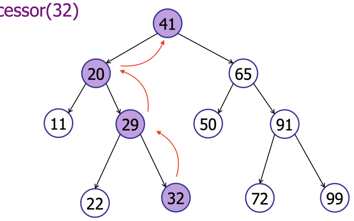
Notes

- Height-balanced $\Rightarrow$ Balanced
- Balanced $\nRightarrow$ Height-balanced
- A height-balanced tree has height $O(2logn) = O(logn)$
- 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**.

Successor

```
if (u.right != null):
    return findMin(u.right)
else:
    p = u.parent
    # find an ancestor that is a left child
    while (p is a right child):
        go up the ancestry
    if (p == null): return null
    else: return parent
```

successor(32)



Case 2: node has no right child.

Rank & Select

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. Select is the reverse of rank. Given a rank, return the value of the node with that rank.

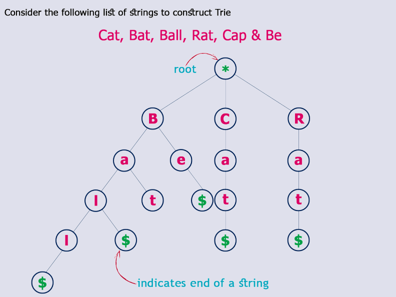
```
# computes the rank of `u` within the subtree
rooted at `root`
getRank(u, root):
    if (u.key < root.key):
        return getRank(u, root.left)
    else if (u.key > root.key):
        return root.left.weight + 1 +
        getRank(u, root.right)
    else:
        return root.left.weight + 1

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
```

Tries

Tries are useful for partial string operations:

- Prefix queries (find all words that start with 'pi')
- Longest prefix (find the longest word that is a prefix to 'pickled')
- Regex (find all words of the form 'pi??le')



Tradeoffs: Tries vs BST

1. Time

- $O(L)$ vs $O(h * L)$
- Trie operations do not depend on text size or number of words

2. Space

- Tries tend to use more space
- Both use $O(textsize)$ space, but tries have more nodes and thus more overhead
- Array implementations waste space in storing children

Rotations

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

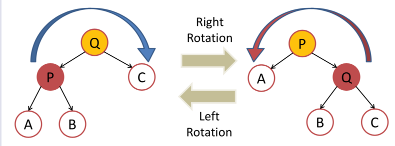
1. If v is out of balance and left heavy:

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

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

(c) `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)`



Notes

- Right rotations require a left child, left rotations require a right child
- The maximum number of rotations required upon insertion is 2, while for deletion it is $O(logn)$

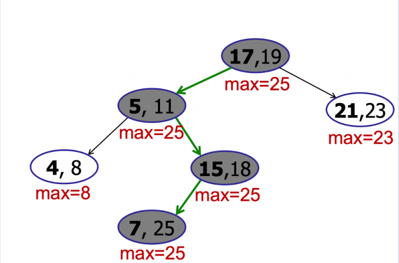
Interval Trees

- AVL Trees with **max value augmentation**
- Goal: `searchInterval(x)` finds an interval that contains `x` in $O(logn)$ time
- Each node stores an interval
- Sorted by key - left endpoint of the interval
- Nodes augmented by **max right-endpoint**

```
# searches in root to find an interval
containing x
def searchInterval(x, root):
    # base cases
    if (root == null): return null;
    if (u.interval.contains(x)): return u

    if (u.left == null || x > u.left.max):
        return searchInterval(x, root.right)
    else:
        return searchInterval(x, root.left)
```

Searching: interval-search(22)



1D Range Searching

Goal: Find keys within `<some interval>`

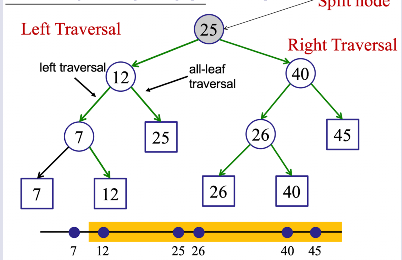
1. Store data in the **leaves** only (build from bottom-up) - $O(nlogn)$
2. Internal nodes store the **max of leaves in left subtree** (Regular BST would work too)
3. Insert/Delete/Rotate don't require modification

```
# find the highest node between l and h
def findSplitNode(l, h, root):
    if (root.key >= h):
        return findSplitNode(l, h, root.left)
    else if (root.key < l):
        return findSplitNode(l, h, root.right)
    else: return root

def leftTraverse(l, root):
    if (l <= root.key):
        # take everything in the right subtree
        enumerateAll(root.right)
        leftTraverse(l, root.left)
    else:
        leftTraverse(l, root.right)

def query(l, h):
    v = findSplitNode(l, h, root) # O(logn)
    leftTraverse(l, v.left) # O(k)
    rightTraverse(h, v.right) # symmetric
```

Example: query(10, 50)



Max Value Augmentation

Each node `u` is augmented with: `value`, that specifies the value associated with that node, and `max`, the maximum `value` of all nodes 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)`

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 update `v.max` `v.right.value)`
- When a node `u` is inserted: Set `u.value = <initial>` and `u.max = <initial>`
- 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` `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`, update `v.max`
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`. Then replace `u` with `v`, and continue to update every node `w` on the path from `v` to the `root`.

- Perform rotations to rebalance.

Total Count 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`

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 + isSpecial(u) ? 1 : 0`

Hashing

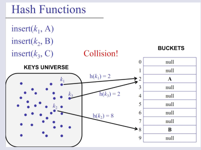
- Goal: insert and search in $O(1)$ time
- Idea: map $n = |U|$ possible keys to m buckets via a hash function
- $h : U \rightarrow \{1 \ldots m\}$
- $time = cost(h) + cost(access)$
- Assume $cost(h) = O(1)$

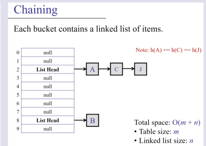
Direct Access Tables

- Maps every possible key to a single bucket i.e. $h : U \rightarrow \{1 \ldots n\}$
- Uses too much space - $O(n)$

Chaining

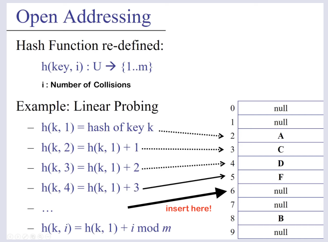
- Maps n possible keys to $m < n$ buckets
- By PHP, $\exists\ h(k_1) = h(k_2)$ i.e. **Collision**
- Uses linked lists to store colliding keys
- Insert - $O(1 + 1)$
- Search - $O(n + 1)$ **(Worst Case)**
- Search - $O(\frac{n}{M} + 1)$ **(SUHA)**
- Optimal Size - $\theta(n)$





Open Addressing e.g. Linear Probing

- If $h(k, 1)$ hits a cluster, the cluster gets larger
- If table is $\frac{1}{4}$ full, there are clusters of size $\theta(logn)$
- In practice, still fast due to caching of nearby memory locations
- Avoid clusters by choosing h that satsifies UHA
- When table is full, cannot insert and search is slow



Desirable Properties for Hash Functions

1. h enumerates all possible buckets
2. Keys are equally likely to be mapped to any permutation (UHA)
3. Keys are equally likely to be mapped to any bucket (SUHA)

- $P(h(k) = m) = \frac{1}{M} = P(collision)$
- $E(collisions) = \frac{n}{M} = LoadFactor(\alpha)$