ORDERS OF GROWTH

Definitions

$T(n) = O(F(n))$ if

$\exists c, n_0 > 0$, for all $n > n_0, T(n) \le cf(n)$

$T(n) = \Omega(F(n))$ if

$\exists c, n_0 > 0$, for all $n > n_0, T(n) \ge cf(n)$

$T(n) = \theta(F(n))$ iff

$T(n) = O(f(n))$ *and* $T(n) = \Omega(f(n))$

Properties

Let $T(n) = O(f(n))$ and $S(n) = O(g(n))$

1. $T(n) + S(n) = O(f(n) + g(n))$
2. $T(n) \cdot S(n) = O(f(n) \cdot g(n))$
3. $T(S(n)) = O(f(g(n)))$
4. Cost of if/else statements: $\max(c1, c2) \le c1 + c2$
5. $\max(T(n), S(n)) \le T(n) + S(n)$

Notes

1. $\sqrt{n} \log n$ is $O(n)$
2. $O(2^n) \neq O(2^{2n})$(*degree matters*)
3. $O(\log(n!)) = O(n \log n)$ →Sterling's approx.
4. $T(n-1) + T(n-2) \ldots = 2T(n-1)$
5. $\sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2}$
6. $\sum_{i=1}^{n} a_i = a_1 + a_2 \ldots + a_n = \frac{n(a_n + a_1)}{2}$
7. $\sum_{i=1}^{n} 2^i = 2^1 + 2^2 \ldots + 2^n = 2^{n+1} - 1$
8. $\sum_{i=1}^{n} a_i = a_1 + a_2 \ldots + a_n = a_1 \frac{c^n - 1}{c - 1}, if\ a_n = ca_{n-1}$
9. $\sum_{i=1}^{\infty} a_i = \frac{a_1}{1-c}, if\ 0 < c < 1$
10. $\sum_{i=1}^{n} \frac{1}{i} = 1 + \frac{1}{2} + \cdots \frac{1}{n} \approx \ln(i+1)$
11. $\sum_{i=1}^{n} i^2 = 1^2 + 2^2 \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$

Master Theorem

$T(n) = aT\left(\frac{n}{b}\right) + f(n)$ **where** $a \ge 0, b > 1$

$\theta(n^{\log_b a})$ → $f(n) < n^{\log_b a}$ *polynomially*

$\theta(n^{\log_b a} \log n)$ → $f(n) = n^{\log_b a}$

$\theta(f(n))$ → $f(n) > n^{\log_b a}$ *polynomially*

SORTING ALGO	Description	Invariant
Bubble Sort	Compare adjacent (1st 2nd,2nd 3rd) items and swap.	Largest i elements are sorted
Selection Sort	Select minimum element from range of low/high index, swaps into position. Repeat with increasing low index until all elements have been selected. Has least swaps needed	Smallest i elements are sorted
Insertion Sort	Compare the key with the previous elements. If the previous elements are greater than key, swap key to left until it is smaller. Start from index 1 to array size.	Subarray A[0 to i-1] is always sorted
Heap Sort	Repeatedly extractMax() element from heap, place it at end of array, update heap	In max heap, last i elements are sorted, vice versa min heap
Merge Sort	Divide array into half, recursively sort, then merge	Each subarray is already sorted when merging
Quick Sort	Partition around chosen/random element. Low/high index move left/right until element is bigger/smaller than pivot, swap low and high, then repeat on subarrays.	All elements to the left/right of pivot are smaller/larger. Partition is in right position

Quick Sort

Partition algorithm: O(n)

Stable quicksort: O(log n) space

> 1st element as partition, 2 pointers from left to right

- left pointer moves until index > pivot
- right pointer moves until index < pivot
- swap elements until left = right

> swap partition and left=right index

Quick Sort Optimizations

1. 3-way partition w/ dup array $O(nlogn), O(n^2)$ w/o dup array

- 4 pointers - in-progress, < pivot, = pivot and > pivot

> If A[i] < pivot -> swap in-progress pointer with < pivot pointer

> If A[i] = pivot -> swap in-progress pointer with = pivot pointer

> If A[i] > pivot -> swap in-progress pointer with > pivot pointer

2. Stable if partitioning is also stable
3. Extra memory for stable quick sort

Choice of Pivot

1. $O(n^2)$: 1st/last/middle element
2. $O(n \log n)$: median/random element

> same if split by fractions

3. Choose at random- random var runtime

Quick Select

O(n) to find the kth smallest element

1. After partition, pivot is always in correct position
2. Recurse left/right of pivot if kth is smaller/bigger.

Duplicates works on quick select.

ALGO	Best	Average	Worst	Stable
Bubble	$\Omega(n)$	$O(n^2)$	$O(n^2)$	✓
Selection	$\Omega(n^2)$	$O(n^2)$	$O(n^2)$	✕
Insertion	$\Omega(n)$	$O(n^2)$	$O(n^2)$	✓
Heap	$\Omega(n \log n)$	$O(n \log n)$	$O(n \log n)$	✕
Merge	$\Omega(n \log n)$	$O(n \log n)$	$O(n \log n)$	✓
Quick	$\Omega(n \log n)$	$O(n \log n)$	$O(n^2)$	✕
Quick Select	$O(1)$	$O(n)$	$O(n^2)$	✕

data structures assuming $O(1)$ comparison cost

data structure	search	insert
sorted array	$O(\log n)$	$O(n)$
unsorted array	$O(n)$	$O(1)$
linked list	$O(n)$	$O(1)$
tree (kd/(a, b)/binary)	$O(\log n)$ or $O(h)$	$O(\log n)$ or $O(h)$
trie	$O(L)$	$O(L)$
dictionary	$O(\log n)$	$O(\log n)$
symbol table	$O(1)$	$O(1)$
chaining	$O(n)$	$O(1)$
open addressing	$\frac{1}{1-\alpha} = O(1)$	$O(1)$

orders of growth

$T(n) = 2T(\frac{n}{2}) + O(n) \Rightarrow O(n \log n)$

$T(n) = T(\frac{n}{2}) + O(n) \Rightarrow O(n)$

$T(n) = 2T(\frac{n}{2}) + O(1) \Rightarrow O(n)$

$T(n) = T(\frac{n}{2}) + O(1) \Rightarrow O(\log n)$

$T(n) = 2T(n-1) + O(1) \Rightarrow O(2^n)$

$T(n) = 2T(\frac{n}{2}) + O(n \log n) \Rightarrow O(n(\log n)^2)$

$T(n) = 2T(\frac{n}{4}) + O(1) \Rightarrow O(\sqrt{n})$

$T(n) = T(n-c) + O(n) \Rightarrow O(n^2)$

searching

search	average
linear	$O(n)$
binary	$O(\log n)$
quickSelect	$O(n)$
interval	$O(\log n)$
all-overlaps	$O(k \log n)$
1D range	$O(k + \log n)$
2D range	$O(k + \log^2 n)$

orders of growth

$$1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < 2^n < 2^{2n}$$
$$\log_a n < n^a < a^n < n! < n^n$$

TREES

Binary Search Tree (BST)

1. Either empty, or node pointing to 2 BST
2. Tree balance depends on insertion order
3. Balanced tree: $O(h) = O(\log n)$
4. For full tree of size $n, \exists k \in +int, n = 2^k - 1$

BST Operations

1. `height(n) = max(height(n.left), height(n.right))`
2. `search(n)`, `insert(n)` : $O(h)$
3. `delete(n)` : $O(h)$
   1. no children – remove node
   2. 1 child – remove node, connect parent to child
   3. 2 children – delete successor, replace node with successor.
   4. `searchMin(n)`: $O(h)$ recurse left tree

5. `successor(n)`: $O(h)$
  1. If node has right subtree, `searchMin(n.right)` else: traverse upwards, return 1st parent that contains key in left subtree.

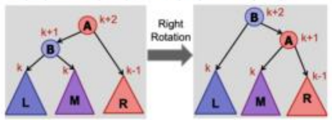
AVL Trees

1. Height-balanced iff |`n.left.height - n.right.height`| ≤ 1
2. Node is augmented with its height
3. Space complexity: $O(LN)$ for N strings of length L

Rebalancing

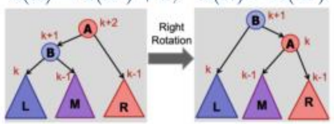
[case 1] B is **balanced: right-rotate**

$h(L) = h(M), \quad h(R) = h(M) - 1$



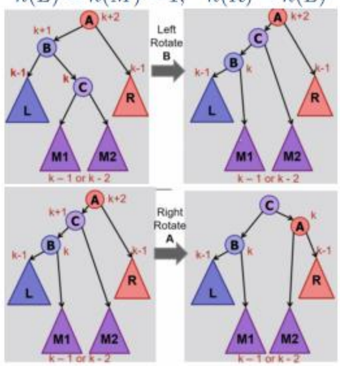
[case 2] B is **left-heavy: right-rotate**

$h(L) = h(M) + 1, \quad h(R) = h(M)$

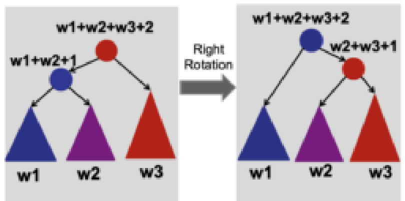


[case 3] B is **right-heavy: left-rotate(v.left), right-rotate(v)**

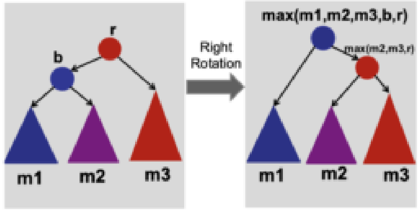
$h(L) = h(M) - 1, \quad h(R) = h(L)$

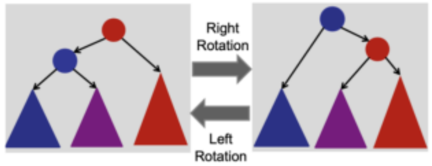


Updating nodes after Rotation
weights



max





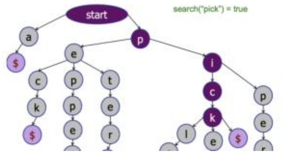
1. Worst case Insertion: max 2 rotations
2. Worst case Deletion: O(log n)
3. Rotations can create every tree shape

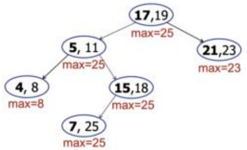
Tries

1. `search(n)`, `insert(n)`: $O(L)$
2. Space: $O(\textit{text size} - \textit{overhead})$

Interval Trees

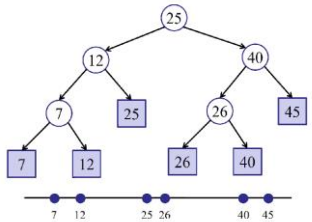
1. `search(key)`: $O(\log n)$
   > if value in root interval, return
   > if value > max(left subtree), recurse right
   > else recurse left (only when you can't go right)
2. All overlaps: $O(k \log n)$ for k overlapping intervals (repeat algo until no more intervals)





Orthogonal Range Searching

1. BST: leaves store points, parent nodes store max value in left subtree



2. `build(points[])`: $O(n \log n)$
3. `query(a, b)`: $O(k + logn)$ for k points

1. Find Node between low/high, starting at root =`findSplit(low,high)`: $O(n \log n)$
2. Output all node in right subtree & recurse left, or recurse right = `leftTraverse(n)`: $O(k)$
3. Symmetric to left traversal `rightTraverse(n)`: $O(k)$
4. `insert(key)`: $O(\log n)$
5. `nodeCount(v,low,high)`: Left-traverse but count weight of right subtree instead of traversing.

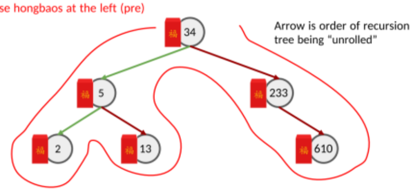
Order Statistics

1. Finding rank k in an augmented AVL
   1. Rank = left.weight + 1
   2. If k is ranked, return node
   3. If k<rank, recurse left subtree with rank, else recurse right subtree with rank-1
2. Find rank given node n
   1. If n has left child, rank = left.weight + 1
   2. Else set node as rank = 1, traverse upwards
   3. Go to parent, if node is parent's left child, keep rank
   4. If node is right child, rank += parent.left.weight +1, continue traversal upwards
3. Maintain weight during insertions
   > Add item, then traverse upwards and add 1 to each node until root is reached
   > If tree is not balanced, rotate to balance
     - When doing right rotation, only need to update the root and parent node for weight

Tree Traversal

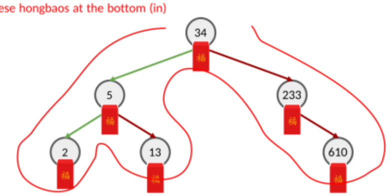
Pre Order: print, root , left, right

Lifehack: Put these hongbaos at the left (pre)



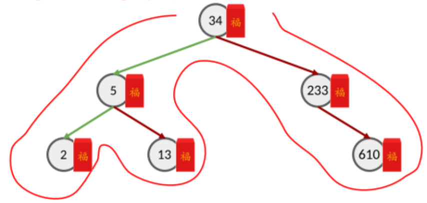
In Order: left, print, root, , right

Lifehack: Put these hongbaos at the bottom (in)



Post Order: left, right, print, root

Lifehack: Put these hongbaos at the right (post)



KD Tree

1. Median using quick select as root, alternates splitting via x and y coordinates
2. `construct(points[])`: $O(\log n)$
3. `search(points)`: $O(h)$
4. `minimum(points)`: $2T\left(\frac{n}{4}\right) + O(1) = O(\sqrt{n})$

Priority Queue

Data	Insert	ExtractMax
Sorted Array	$O(n)$	$O(1)$
Unsorted	$O(1)$	$O(\log n)$
AVL Tree	$O(\log n)$	$O(\log n)$

Heap

1. Max heap: Stores biggest in root, smallest in leaf
   Min heap: Stores smallest in root, biggest in leaf
2. Priority of parent always >= child in max heap
3. It is a complete binary tree: every level is full (has both left/right child) | all leaves are far left
4. `height(n)= floor(log n)`
5. `insert(n)`: far left if priority of n > parent: Bubbleup/swap. Check 2. And 3.
6. `extractMax(n)`: return root and delete root
7. `increase(n,a)`: increase n to a, Bubbleup a
8. `decrease(n,b)`: decrease n to b, move b down, bubbleDown to child with bigger priority
9. `delete(n)`: swap n with last(), remove last(). bubbleDown /up depending on heap order

Mapping heap into array:

1. Start from root, at each level, insert from left/right
2. Insert: insert into empty array slot, bubble up by swapping indexes
3. Left(x) = $2x + 1$ | Right(x) = $2x + 2$
4. Parent(x) = $floor(\frac{x-1}{2})$

Mapping unsorted array into heap:

1. Iterate from end of array, bubbleDown current index and array: $O(n)$