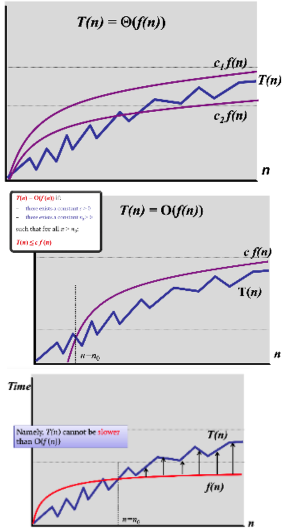
ORDERS OF GROWTH

definitions



$T(n) = \Theta(f(n)) \leftrightarrow T(n) = O(f(n)) \wedge T(n) = \Omega(f(n))$

$T(n) = O(f(n))$ if $\exists c, n_0 > 0$ s.t. $\forall n > n_0, T(n) \le cf(n)$

$T(n) = \Omega(f(n))$ if $\exists c, n_0 > 0$ s.t. $\forall n > n_0, T(n) \ge cf(n)$

properties

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

- addition: $T(n) + S(n) = O(f(n) + g(n))$
- multiplication: $T(n) * S(n) = O(f(n) * g(n))$
- composition: $f1 \circ f2 = O(g1 \circ g2)$
  - only if both functions are increasing
- if/else: $cost = max(c1, c2) \le c1 + c2$
- max: $max(f(n), g(n)) \le f(n) + g(n)$

space complexity

Assumption: Exiting function release mem

SORTING

insertion sort faster than other O(n^2) algos

invariants

selection sort – Smallest j element sorted
bubble sort – Largest j elements sorted
insertion sort – First j elements sorted
merge sort – Subarrays are sorted; $O(n)$ merging
quick sort – Pivot is in sorted position

quicksort

partition takes $O(n)$ time
array of duplicates takes $O(n^2)$ without 3-way

inplace algo

$O(\log n)$ space

- First element as partition
- While left != right
  - Increment left until element > pivot
  - Decrement right until element < pivot
  - <Break cond here>
  - Swap left and right elements
- Swap partition with right index

3-way inplace algo

- Iterate through array and maintain left right
  - If current < pivot, swap start and current, increment current and left by 1
  - If current > pivot, swap end and current, decrement right by 1 but not current
  - If current = pivot, increment current

quick select

$O(n)$ average to find k^{th} smallest element
Invariant: after partition, partition correct position

TREES

BST

Height of leaf nodes = 0
Height of balanced tree = $\log_2 n$
Deletion – If 2 children, delete successor and replace subtree root with successor value
Finding successor: find min of right subtree or traverse upwards and find first left parent

Scapegoat Trees

A tree that is not α-height-balanced is not α-weight-balanced. Scapegoat trees are not guaranteed to be α-weight-balance, but are loosely α-height-balanced

AVL

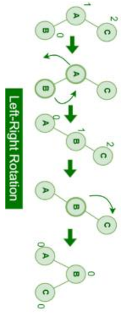
`|v.left.height - v.right.height| <= 1`

height and nodes

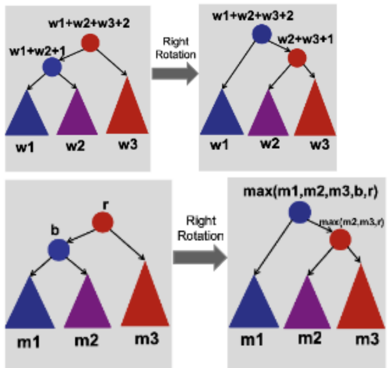
Min height given n nodes: $floor(\log_2 n)$
Max height given n nodes: $1.44 * \log_2 n$
Max nodes given h height: $2^{h+1} - 1$
Min nodes given h: $N(h) = N(h-1) + N(h-2) + 1$ for $n>2$
where $N(0) = 1$ and $N(1) = 2$

insertion rebalancing

left-left – Right rotate unbalanced node
right-right – Left rotate unbalanced node
left-right – L rot child, R rot unbalanced
right-left – R rot child, L rot balanced



updating nodes after rotation



deletion

recurse upwards and rebalance every node

tries / prefix tree

$O(L)$ time for search/insert
$O(Nk)$ space – N nodes * k overhead for pointers

order statistics tree (rank)

Augmented AVL tree

select(k)

```
rank = m_left.weight + 1;
if (k == rank) then
    return v;
else if (k < rank) then
    return m_left.select(k);
else if (k > rank) then
    return m_right.select(k-rank);
```

rank(node)

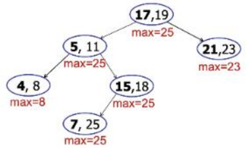
```
rank = node.left.weight + 1;
while (node != null) do
    if node is left child then
        do nothing
    else if node is right child then
        rank += node.parent.left.weight + 1;
    node = node.parent;
return rank;
```

interval trees

sort by interval start/min, store max of subtree

interval-search(x)

```
c = root;
while (c != null and x is not in c.interval) do
    if (c.left == null) then
        c = c.right;
    else if (x > c.left.max) then
        c = c.right;
    else c = c.left;
return c.interval;
```



All-Overlaps Algorithm:

Repeat until no more intervals:
– Search for interval.
– Add to list.
– Delete interval.

Repeat for all intervals on list:
– Add interval back to tree.

(a, b)–Trees

rules

1. Internal nodes have a-b children where: $2 <= a <= (b+1)/2$

node type	# keys min	# keys max	# children min	# children max
root	1	$b-1$	2	b
internal	$a-1$	$b-1$	a	b
leaf	$a-1$	$b-1$	0	0

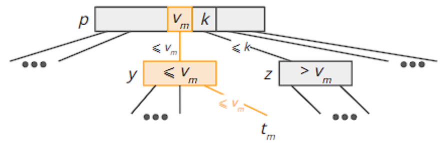
2. Non-leaf must have 1 more child than number of keys
3. All leaves are the same depth

operations

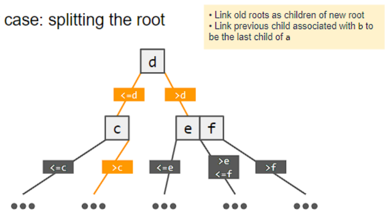
height – Min $O(\log_b n)$, Max $O(\log_a n)$
search – $O(\log n)$; $O(\log_2 b * \log_a n)$ to binary search at each node
insert - $O(\log n)$
When inserting, check for too many children and *split* if children more than b

split

1. Pick the median key v_m as the split key
2. Split z into LHS and RHS using v_m
3. Create a new node y
4. Move LHS split from z to y
5. Create a new empty node r
6. Insert v_m into r
7. Promote r to new root node
8. Assign y and z to be the left and right child of r respectively
9. Assign previous subtree t_m associated with v_m to be the final child of y

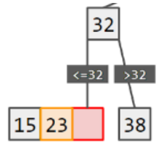


Corner case: splitting the root



deletion – $O(\log n)$
If node becomes empty, merge with sibling
After merge, may be too large; Use **share** operation instead (merge then split)
`len(w + z) >= b-1 ? share(w,z) : merge(w,z)`

merge
Reverse of split: Use parent node to join 2 children



B–Tree

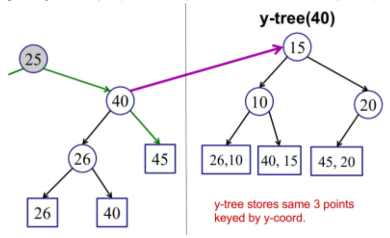
(a,b)-tree with a = B, b = 2B

Orthogonal Range Searching

Binary Tree for coordinate points ($O(n)$ space)
Internal nodes: $max(node.left)$; Leaves: points
Invariant – Search interval for left-traversal at node v includes maximum item in the subtree v
query – Find all points from a to b; $O(k + \log n)$ for k pts
Steps: Find split node, recursively output left/right
insert – $O(\log n)$, **build tree** – $O(n \log n)$

2D-Range Tree Variant

Store a y-tree at each x-node ($O(n \log n)$ space)
query – $O(\log^2 n + k)$, **build tree** – $O(n \log n)$



Space: O(n $\log^{d-1}$n)
buildTree cost: O(n $\log^{d-1}$n)
Query cost: O($\log^d$n + k)

Cannot insert/delete because of $O(n)$ to rotate

Qn 1 Identifying sorts
Refer to invariants
Quicksort: First element as pivot, see which already has initial first element (pivot) in correct place
Bubble: Largest sorted at the end; In place - Remaining ordering is not affected
Insertion: First k sorted, not smallest, remaining untouched
Selection: Smallest at start; Initial start must be swapped down
Merge: Groups of orders of 2 should be sorted

data structures assuming $O(1)$ comparison cost

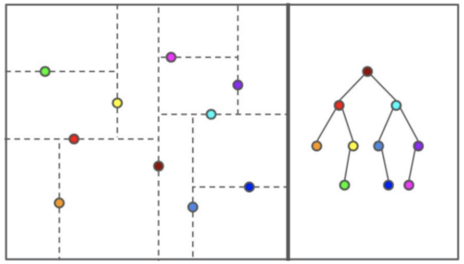
data structure	search	insert
sorted array	$O(\log n)$	$O(n)$
unsorted array	$O(n)$	$O(1)$
linked list	$O(n)$	$O(1)$
tree (kd/(a, b)/binary)	$O(\log n)$ or $O(h)$	$O(\log n)$ or $O(h)$
trie	$O(L)$	$O(L)$

searching

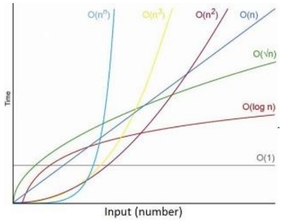
search	average
linear	$O(n)$
binary	$O(\log n)$
quickSelect	$O(n)$
interval	$O(\log n)$
all-overlaps	$O(k \log n)$
1D range	$O(k + \log n)$
2D range	$O(k + \log^2 n)$

kd–tree

Partition tree using nodes; Alternate x-y
construct – quick select median $O(n)$; $O(n \log n)$ total
min/max(x) – recurse both sides for y checks; $O(\sqrt{n})$
When split by x, go down the appropriate subtree.
When split by y, check both. $T(n) = 2T\left(\frac{n}{4}\right) + O(1)$
search – $O(h)$



```python
def binary_search(arr, target):
    low, high = 0, len(arr) - 1
    while low <= high:
        mid = (low + high) // 2
        mid_element = arr[mid]
        if mid_element == target:
            return mid
        elif mid_element < target:
            low = mid + 1
        else:
            high = mid - 1
    return -1
```



sort	best	average	worst	stable?	memory
bubble	$\Omega(n)$	$O(n^2)$	$O(n^2)$	✓	$O(1)$
selection	$\Omega(n^2)$	$O(n^2)$	$O(n^2)$	×	$O(1)$
insertion	$\Omega(n)$	$O(n^2)$	$O(n^2)$	✓	$O(1)$
merge	$\Omega(n \log n)$	$O(n \log n)$	$O(n \log n)$	✓	$O(n)$
quick	$\Omega(n \log n)$	$O(n \log n)$	$O(n^2)$	×	$O(1)$

Compiled by @jloh02
Adapted from https://github.com/jovyntls/cheatsheets

Simplifying Recurrence Relations & Order of Growth

Orders of growth:

- $1 < \log n < \sqrt{n} < n < n \log n < n^2 < n^3 < 2^n < 2^{2n}$
- $\log_a n < n^a < a^n < n! < n^n$
- $\sqrt{n} \log n \to O(n)$
- $O(2^{2n}) \neq O(2^n)$

Math Formulas
Logarithmic: $a^{\log b} = b^{\log a}$

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

$\log n^c = c \log n$

$\log ab = \log a + \log b$

AP/GP Sums:

$$S_n = \sum_{i=0}^{n-1} ar^i = a\left(\frac{1-r^n}{1-r}\right)$$

$$S_n = \sum_{i=0}^{n-1} (a + id) = \frac{n}{2}(2a + (n-1)d)$$

Harmonic Numbers:

$$\sum_{i=0}^{n} \frac{1}{i} = \Theta(\log n)$$

Stirling's Approximation:

$$\sum_{i=1}^{n} \log i = \log(n!) = \Theta(n \log n)$$

Interpreting different time complexities
$O(g(n)) \to f(n) <= g(n)$
$\Theta(g(n)) \to f(n) = g(n)$
$\Omega(g(n)) \to f(n) >= g(n)$

Solving recurrence relations
For T(n) = a(T(n/b) + f(n)
Use pattern and assume n = b^k
Reduce until T(1)
Use sum of gp formula to reduce

Master's Theorem
Theorem 1 *The recurrence*

$$\begin{aligned} T(n) &= aT(n/b) + cn^k \\ T(1) &= c, \end{aligned}$$

where a, b, c, and k are all constants, solves to:

$$\begin{aligned} T(n) &\in \Theta(n^k) \text{ if } a < b^k \\ T(n) &\in \Theta(n^k \log n) \text{ if } a = b^k \\ T(n) &\in \Theta(n^{\log_b a}) \text{ if } a > b^k \end{aligned}$$

Examples

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

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

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

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

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

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

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

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