Time Complexity

1 < logn < sqrt(n) < n < nlogn = logn! < n^2 < 2^n

Recurrence	Runtime
T(n) = 2T(n-1) + O(1)	2^n
T(n) = 2T(n/2) + O(n)	nlogn
T(n) = T(n/2) + O(n)	n
T(n) = 2T(n/2) + O(1)	n
T(n) = S(0,sqrt(n))T(i) + sqrt(n)	n
T(n) = 2T(n/4) + O(1)	sqrt(n)
T(n) = T(n/2) + O(1)	logn

Searching

Search	Runtime
Linear	n
Binary	logn
Quickselect	n
Knuth Shuffle	n
Merkel Tree	logn

Knuth Shuffle:
for (1,n-1): swap(i,rand(0,i))

Sorting

Algorithm	Best	Average	Worst	Stable	Invariant
Bubble	n	n^2	n^2	Yes	Largest k items are in final k positions
Selection	n^2	n^2	n^2	No	Smallest k items are in smallest k positions
Insertion	n	n^2	n^2	Yes	First k items are sorted
Merge	nlogn	nlogn	nlogn	Yes	Groups of 2^x are sorted
Quick	nlogn nlogk	nlogn nlogk	n^2 nk	No	Array is partitioned around pivot T(n) = kT(n/k) + O(nlogk) -> nlogn
Reversal		n(logn)^2			Quicksort with Mergesort around pivot

Trees

Structure	Search	Insert	Delete	Remark
Binary Search	h	h	h	**Delete** if x has 2 child: replace x with successor(x) **Successor** right.min() or recurse to (left of parent or root)

AVL	logn	logn+2R	logn+lognR	h < 2logn or n > 2^(h/2) v.left Left Heavy or Balanced: right(v) v.left Right Heavy: left(v.left), right(v)
Trie	L	L	L	More space due to more overhead
(a,b)	logn	logn	logn	**split** for insert, **merge+share** for delete
kd	h	h	h	Alternate splitting horizontally and vertically

Augmented Structure	Remark
Dynamic Order Statistic	Stores weight of subtree. During functions, rank = left.weight + 1 **Select** left.weight < rank: left.select(k). Else: right.select(k-rank) **Rank** recurse to root, if node is right child: rank += parent.left.weight + 1
Interval Tree	Sort by left endpoint. Stores max endpoint in node's subtree **Search** (logn) If x > max or left is null, search(right). Else: search(left) **All Overlap** (klogn) search node, add to list, delete node, repeat until null
Orthogonal Range Search	Store all points as leaves of a BST. Internal nodes stores max of left. **Range Query** (k+logn) find split node. do left & right traversals. **2D Range Query** (k+(logn)^2) for node in x-tree, build y-tree using nodes in subtree.

Hashing

Must redefine **hashCode** *default returns address* and **equals** *for **get** to work*

Collision	Insert	Search	Space	Remarks
Linked List	h+1 = 1	h+n/m = 1	m+n	Simple Uniform Hashing Assumption Worst case for search = n
Open Addressing	1/(1-a),	where a = n/m a< 1	n	Uniform Hashing Assumption Linear Probing - Clusters Double Hashing - h(k,i) = f(k) + ig(k) mod *m*, (m,g(k)) **Delete** sets node to special value for **search**