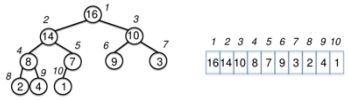
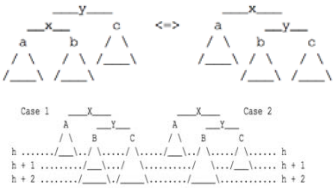
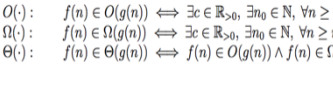
Heaps (finding min/max in constant time)	BST (sorting in nlogn)	AVL (sorting in nlogn and want balance)
-Binary trees that satisfy **max-heap property**: each node must be >= children. It must also have depth within 1 of each other (i.e. must be balanced) and must be left justified. -Can sort in place but is not stable -Max(H) - return max node O(1) -Max-heapify(H) - fix subtree rooted at x - O(logn) To implement: Find the maximum of the two children of x, and swap with x if larger. recursively call MAX-HEAPIFY on the subtree rooted at x. The runtime is proportional to the height of the tree, since we "trickle down" the whole tree but heapify works bottom up - extract-max(H): remove max val in H and return its value - O(logn). To implement:swap root of H with last leaf in array. Remove last leaf = max. Call max-heapify on root. -increase-key(H,x,k) - increase x to k, O(logn). To implement: increase key and then trickle up - swap with parent as long as parent smaller. -insert(H,k): add leaf with -infinity than run increase to k - O(logn) -delete: increase to infinity, extract map -check max heap: check heap property, check if depth of leaf is d, if not it should be d-1 and stay that way  Children - 2i, 2i+1, parent = (i)/2 floor. Or x/2 floor, 2x, 2x+1. -build-heap: put it in arbitrarily. Assume the floor n/2 leaves, heapified already, call max heapify from bottom up. **O(n)** -heap-sort: **O(nlogn)**, build max heap, extract max repeatedly, *in place*, not stable. - at each level, *l*, you have 2^*l* leaves, count level from 0. At each level you have 2^(*l*+1)-1 total nodes. Total number of nodes in heap is O(2^h), h = O(logn). Maximum number of elements in heap = 2^(h+1) - 1= sum of 2^i from i =0 to h. Min number = 2^h.	-Array size is O(2^h). -BST Property: left subtree <= x <= right subtree. **-stores x.key, x.left, x.right, x.parent, height** -dynamic data structure -When given BST, and want to check if BST in O(n) do by in-order traversal. Can do *iff* the tree is a BST. Traverse and output each key in order of tree (O(n)), and check it's in increasing order (O(n)). Can **traverse in O(n)** because each edge traversed upwards at most once and downwards at most once -it is not enough to check left, right; must check min max of subtrees: X.left.max <= x <= x.right.min -creating a tree worst case n^2, **expected O(nlogn) to create** -find-min: walk left -find: walk, check right and left -find-next: if self.right -> self.right.find_min. Else: node = self.parent. While node and node.key < self.key, node = node.parent. Continue until node.key > self.key. **-insert: O(h)** traverse tree downwards while maintaining invariant until find empty location - **takes O(nlogn) to build** -find predecessor/successor in O(logn) -left child index = 2i+1, right = 2i+2 **-delete:** 3 cases- no children (just delete), one child (link x's parent to x's child, and cut out x), two children (replace x with the larger child)	-Max height of O(2logn) but height is always O(logn) = A BST on n keys that support normal BST operations but guarantees height of tree is always O(logn) -you can create an AVL tree from a sorted list of numbers in O(n) time. Pick median as root. -**Creating a tree is O(nlogn), sorting is also O(nlogn) from that tree and the sort will be stable but not in place** -AVL property: For every node in an AVL tree, the magnitude of skew is at most one height = max(left_h, right_h) + 1 skew = right_h - left_h To check if AVL, if skew > 1, return false; return max(left_h,right_h) + 1 -rebalancing: an inserted node can't have children so it satisfies AVL. Rebalance all nodes along root to leaf path. Parent of a deleted node could have a child, but child's height not affected by deletion. So we rebalance ancestors. Rebalance root to leaf - rebalance all ancestors of the leaf. - always rotate lowest unbalanced node -successor is not next or prev; it takes in a key which needs to be found - search for key and maintain largest key seen so far - only ever need O(n) rotations -  

Insertion Sort (use if have low space constraints)
In place algorithm; O(1) extra space. for i = 1, 2, ..., n − 1: insert element A[i] into the sorted array A[0 : i] by pairwise swaps down to its leftmost position**. Runtime of this algorithm is O(n^2). It is stable.**

Merge Sort (use when want element to maintain same position in sorting, sort linked lists: divide and conquer algorithm)
If the length of the list is 1, the list is sorted. Return the list. Otherwise, split the list in two (roughly) equal halves and then recursively merge sort the two halves. Merge the two sorted halves into one sorted list. **Runtime worst-case O(nlogn), need O(n) space to merge, it is stable. Merging takes O(L+R).**

Counting Sort (use when want to sort in linear time: sort in O(n+k) time when elements are in range from 1 to k.)
Takes an array A of n elements with keys in the range {1, 2, ..., k} and **sorts the array in O(n + k) time.** It is a ***stable sort*** if you have array C where the value of C[j] is the number of elements ≤ j in A. Now, we iterate through A backwards starting from the last element of A. For each element i, check C[i] to find out how many elements are there ≤ i. We decrement C[i] so that if we see a duplicate element.

Radix Sort (comparison based sorting in Ω(nlogn) time: use for sorting integers - no decimals or repeating digits) If have decimals, multiply by factor of 10 to get rid.
Essentially need a finite number of bins to sort items into. If there are *n words* of *length d* in an *alphabet/base of size b*, then each counting sort takes time Θ(n + b), for a **total time of Θ((n + b)d).** With finite number of length and b takes O(n) time. If you have [-n^100, n^100], treat it as base n integer with 101 digits. Or if you have 0 <=x <= n^3 x is a 3 digit base n number. Not in place bc uses buckets to hold data. **If largest integer in the set u<= n^c, then radix sort runs in O(nc) time. If c is constant, then radix sort runs in linear time.**

Direct access arrays: normal static array that associates a semantic meaning with each array index location: any item x with key k will be stored at array index k.
Use when we want to store a set of n items, each associated with a unique integer key in the bounded range (0 : u-1)
Items will be stored in a length u direct access array where each array slot i contains an item associated with integer key i if it exists. To find an item having integer key i, a search algorithm simply looks in array slot i to respond to the search query in worst-case constant time. BUT order operations on DAA very slow (no idea where first, last, or next elements are). Constant time lookup comes at cost of storage space: DAA must have a slot available for every possible key in range.

Hashing: Want the performance benefits of a DAA while only using linear O(n) space when n <<u
Store items in a smaller dynamic direct access array with m = O(n) slots instead of u. Therefore, array grows and shrinks depending on #items stored. Hash function/map: *h(k) : {0, …, u-1} —> {0, …, m-1}*. Universal hashing: choose hash function randomly from a large family of hash functions that are independent of the input. Hash table: smaller direct access array. When the hash function is injective (no two keys map to the same DAA index), support worst-case constant time search. When m < u, by pigeonhole principle hash function would map multiple keys to the same array index = collision (with universal hashing, probability of collision is 1/m). **Search(x): expected O(1), worst-case O(n) when all chained to one index**
Probability(collision) = $(1-\frac{1}{m})^{n-1}$ **Expected # empty slots= m(**$\frac{m-1}{m}$**)**^{n}
****Insertion takes Θ(n) time ****Initializing and scanning the DAA takes Θ(u) time —> Algorithm runs in Θ(n+u) time. If u = O(n), then algorithm is linear.**
Chaining: collision resolution strategy where colliding keys are stored separately from the original hash table. Each hash table index holds a pointer to a chain, separate data structure that supports operations {find, insert, delete} ****Worst-case all keys stored at one index in hash table: then the chain will have linear size, and all operations will take **worst-case linear time**. ****A hash table where collisions are resolved using chaining, implemented using a randomly chosen hash function from a universal family will **perform dynamic set operations in expected constant time.** ****Having to**

rebuild the DAA to a different size, choose a new hash function, and reinsert all items back into the hash table leads to amortized bounds for dynamic operations. ****Cannot handle duplicate keys nor large key ranges.**

		$g(n)$		
		$n^{0.01}$	$n^{\log\log n}$	$n\log_{15} n$
$f(n)$	$n\log n$	Ω	O	Θ
	$(\log n)^{\log n}$	Ω	Θ	Ω
	$2^{\sqrt{\log n}}$	O	O	O
	$n\log\binom{n}{2}$	Ω	O	Θ

Useful observations:

$$n^{0.01} = 2^{0.01\cdot\log n}$$
$$n^{\log\log n} = 2^{\log n\cdot\log\log n}$$
$$(\log n)^{\log n} = 2^{\log n\cdot\log\log n}$$
$$n\log\binom{n}{2} = n\log\frac{n!}{(n-2)!2!} = n\log\frac{n(n-1)}{2} = \Theta(n\log n).$$

Algorithm	Time $O(\cdot)$	In-place?	Stable?	Comments
Insertion Sort	n^2	Y	Y	$O(nk)$ for k-proximate
Selection Sort	n^2	Y	N	$O(n)$ swaps
Merge Sort	$n\lg n$	N	Y	stable, optimal comparison
Heap Sort	$n\lg n$	Y	N	low space, optimal comparison
AVL Sort	$n\lg n$	N	Y	good if also need dynamic
Counting Sort	n	N	Y	$u = \Theta(n)$ is domain of possible keys
Radix Sort	cn	N	Y	$u = \Theta(n^c)$ is domain of possible keys

Priority Queue Data Structure	Operations $O(\cdot)$		
	`build(A)`	`insert(x)`	`delete_max()`
Dynamic Array	n	$1_{(a)}$	n
Sorted Dyn. Array	$n\log n$	n	$1_{(a)}$
Set AVL	$n\log n$	$\log n$	$\log n$
Binary Heap	n	$\log n_{(a)}$	$\log n_{(a)}$

Sequence Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic		
	`build(A)`	`get_at(i)` `set_at(i,x)`	`insert_first(x)` `delete_first()`	`insert_last(x)` `delete_last()`	`insert_at(i, x)` `delete_at(i)`
Array	n	1	n	n	n
Linked List	n	n	1	n	n
Dynamic Array	n	1	n	$1_{(a)}$	n
Sequence AVL	n	$\log n$	$\log n$	$\log n$	$\log n$

Set Data Structure	Operations $O(\cdot)$				
	Container	Static	Dynamic	Order	
	`build(A)`	`find(k)`	`insert(x)` `delete(k)`	`find_min()` `find_max()`	`find_prev(k)` `find_next(k)`
Array	n	n	n	n	n
Sorted Array	$n\log n$	$\log n$	n	1	$\log n$
Direct Access	u	1	1	u	u
Hash Table	$n_{(e)}$	$1_{(e)}$	$1_{(a)(e)}$	n	n
Set AVL	$n\log n$	$\log n$	$\log n$	$\log n$	$\log n$

Augmentations:
- Augmentation has to be "_____ of the subtree" (characteristic, relevant to problem)
- Computable from children's augmentation in O(1) time

Asymptotics:
- If given theta in recurrence, master's theorem will be theta
- If not given omega or O, then just assume omega(1) or O(infinity) for the other case, calculate both

Comparing Fractions:
- When given fraction a/b and c/d:
- Compare a*d and b*c

Radix Sort:
- If longest element in the array a_i< n^d (aka log_n(a_i) < d (constant)), then we can radix sort in O(dn). When d=1, that's counting sort

Master Theorem

If $T(n) = aT(n/b) + O(n^d)$ for constants $a > 0$, $b > 1$, $d \geq 0$, then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a \end{cases}$$

1. Work done at root
2. Work distributed
3. Work done at leaves