

# Data Structures Cheat Sheet

[[Data Structures I](#)] [[Data Structures II](#)] [[Data Structures III](#)] [[Data Structures IV](#)] [**Data Structures Cheat Sheet**]

	add to end	remove from end	insert at middle	remove from middle	Random Access	In-order Access	Search for specific element	Notes
Array	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	Most efficient use of memory; use in cases where data size is fixed.
List<T>	best case $O(1)$ ; worst case $O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	Implementation is optimized for speed. In many cases, List will be the best choice.
Collection<T>	best case $O(1)$ ; worst case $O(n)$	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$	List is a better choice, unless publicly exposed as API.
LinkedList<T>	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(n)$	Many operations are fast, but watch out for cache coherency.
Stack<T>	best case $O(1)$ ; worst case $O(n)$	$O(1)$	N/A	N/A	N/A	N/A	N/A	Shouldn't be selected for performance reasons, but algorithmic ones.
Queue<T>	best case $O(1)$ ; worst case $O(n)$	$O(1)$	N/A	N/A	N/A	N/A	N/A	Shouldn't be selected for performance reasons, but algorithmic ones.
Dictionary<K,T>	best case $O(1)$ ; worst case $O(n)$	$O(1)$	best case $O(1)$ ; worst case $O(n)$	$O(1)$	$O(1)^*$	$O(1)^*$	$O(1)$	Although in-order access time is constant time, it is usually slower than other structures due to the over-head of looking up the key.