ADTs on Parade

ADT	When to use	Array-based?	Linked list-based?	Time complexity: Insertion *Avg case* **Worst case**	Time complexity: Deletion *Avg case* **Worst case**	Time complexity: Search *Avg case* **Worst case**
Array (not an ADT; built-in data structure)	Collection of homogeneous elements.	Yes	No	*O(n)* **O(n)**	*O(n)* **O(n)**	*O(n)* **O(n)** (O(1) to find elmt *i*)
Linked List	Collection of homogeneous elements.	No	Yes	*O(1)* **O(1)**	*O(1)* **O(1)**	*O(n)* **O(n)** (O(n) to find node *i*)
Bag	Growable container. Unordered. Unindexed.	Yes	Yes	*O(n)* [4] **O(n)**	*O(n)* [4] **O(n)**	*O(n)* **O(n)**
Sequence	Growable container. Ordered. Unindexed. Has a current element.	Yes	Yes	*O(n)* [4] **O(n)**	*O(n)* [4] **O(n)**	*O(n)* **O(n)**
Stack	Growable container. Use for LIFO behavior, backtracking, and iterative solutions to recursive problems	Yes	Yes	*O(1)* **O(1)**	*O(1)* **O(1)**	*O(n)* [0] **O(n)**
Queue	Growable container. Use for FIFO behavior, first-come-first-serve simulations	Yes	Yes	*O(1)* **O(1)** [1]	*O(1)* **O(1)** [1]	*O(n)* [0] **O(n)**
Priority queue	Container for retrieving elements in order of priority. Good for scheduling, sorting	Yes	Yes	Depends on how elmts are stored. Usually array or heap-based. See stats for **Array** or **Heap**, respectively.		
Binary search tree	Tree for organizing by a key. Great for all around performance.	Yes [2]	Yes	*O(log n)* **O(n)**	*O(log n)* **O(n)**	*O(log n)* **O(n)**
2-3 tree	Dynamic search tree that maintains balance via node splitting. Better performance than BST at a cost of higher coding complexity. Use to guarantee O(log n) performance at all times.	No	Yes	*O(log n)* **O(log n)**	*O(log n)* **O(log n)**	*O(log n)* **O(log n)**
Heap	Tree where smallest (minheap) or largest (maxheap) item can be obtained in O(1) time. Great for sorting, PQs	Yes	No [3]	*O(log n)* **O(log n)**	*O(log n)* **O(log n)**	*O(n)* **O(n)**
Hash Table	Implementation of the table ADT. A BST-tree-based table or sorted-array-based table are other options.	Yes	No	*O(1)* **O(n)**	*O(1)* **O(n)**	*O(1)* **O(n)**
Graph	Use to establish arbitrary relationships between nodes of data. Good for networks, factories, databases, and many more.	Yes = adjacency matrix	Yes = adjacency list	*O(1)* [5] **O(1)**	Matrix *O(|V|²)* **O(|V|²)**	Adj List [6] *O(|E|)* **O(|E|)**	Matrix [7] *O(1)* **O(1)**	Adj List *O(|V|)* **O(|V|)**

[0] This operation isn't really supported by this ADT.
[1] Assumes wrap around for array-based implementation and firstNode and lastNode pointers for LL-based implementation.
[2] While do-able, an array-based implementation would have poor space complexity if tree not complete.
[3] Well, you could. But no one does. An array provides better performance without the complexity.
[4] These are for the array-based version. The LL-based version would have time complexities listed for LL.
[5] Insertion means inserting a new vertex into the graph (without its edges). |V| is the number of vertices.
[6] Deletion means deleting a vertex (along with its edges). |E| = number of edges, which at worst will be $|V^2|$ if every vertex is adjacent to every other vertex
[7] Here, search refers to searching to see if edge (*x*, *y*) exists