CS3000
Summer 2023
Data Structure Overview

Overview of a few common structures, their basic run-time complexities, and how you might evaluate when and whether they are good choices.

Summary of General Run-Time Complexities

Structure	**Find**	**Insert**	**Delete, Once Found**
Array (unsorted)	O(n)	O(1)	O(n)
Array (sorted)	O(lg n)	O(n)	O(n)
Linked List	O(n)	O(1)	O(1)
Binary Tree	O(n)	O(n)	O(n)
BST (general)	O(h)	O(h)	O(h)
BST (balanced)	O(lg n)	O(lg n)	O(lg n)
Hash Tables	O(1)	O(1)	O(1)

Specialized Structures

Structure	**Insert**	**Remove**
Stack	O(1)	O(1)
Queue	O(1)	O(1)
Heap	O(lg n)	O(lg n)

Graphs

Graph Type	**Add Vertex**	**Add Edge**	**Find Vertex**	**Find Edge**
Adjacency List	O(1) to add to the vertices array	O(1)	O(|V|)	O(|E|), if you know which linked list to look in
Adjacency Matrix	O(1) O(|V|) if we're also concerned with	O(1)	O(|V|)	O(|V|), if you've found one vertex and need to see if it's adjacent to another O(1), if you've found

	initialized all its relevant edges to the null edge			both vertices and know their positions in the matrix

Comparison of Usefulness

Choosing the right data structure for the problem you're trying to solve is rarely straightforward. Sometimes a specific data structure is just right for everything you need; for example, if your algorithm needs the data to respect FIFO order, then it should be pretty obviously to use a queue.

However, it is often the case that many different data structures *could* be used Some will just be able to solve the problem more efficiently. So if there are many possibilities open to you, consider what kinds of operations you'll be needing the most. Consider whether your data will be relatively static, or grow and shrink unpredictably. Consider which one is easier to implement and use. All of these factors can come into play.

Structure	**Good When...**	**Bad When...**
Array	You need random access (indexing) to elements by position. You know the number of elements you'll need. You need to sort your data, once, after it's all been inserted (and then you can do binary search).	After populating the array once, you do a lot of deletes. You need to search for specific elements. You don't know the number of elements you'll need, and you end up over-allocating or needing to expand.
Linked List	Your data dynamically grows and shrinks.	You need random access to elements by position. You want to do binary search, which doesn't work with Linked Lists.
Stack	You need expression evaluation and syntax parsing.	You're interested in elements with a certain value or position
Queue	You need to respect the order in which data was inserted.	You're interested in elements with a certain value or position
Heaps	You need to have easy access to the min or max (O(1) to find it).	You need your data set completely ordered, not partially ordered.
Binary Search Tree	Your data needs to be sorted, and you do a lot of searches. You love recursion.	Your data might be inserted in an ordered way, making the tree unbalanced. Switch to an AVL tree in this case.

	And pointers.	
Graphs	You have a complex “real world” problem to solve, like finding shortest paths.	Your data lends itself to a parent/child structure.
Hash Tables	You want to use an array, but they’re too slow. You want fast access to your data, and you want to be able to delete efficiently.	You need your data to be ordered. Your data + hash function combination result in many collisions.