Linked Lists

▼ Contiguous Lists

A contiguous list refers to a list of entries that is stored in memory cells that are in
sequence with consecutive addresses

▼ Limitations of Contiguous Lists

- We need to determine how much space to allocate to each entry, as well as the number of entries, to determine how much memory to allocate to the entire list
- If the list is a static list that is not going to change, then a contiguous list is a suitable data structure
- However, if the list is a dynamic list which changes frequently, such as when
  entries are added, deleted or rearranged, this would cause a lot of shuffling and
  reallocation of the computer's memory
- In the worst-case scenario, we might add so many entries that the entire list might need to be shifted to a new location in the computer's memory to ensure that there are sufficient contiguous memory cells to store all the entries

▼ Linked Lists

- A linked list is a linear data structure which stores data in nodes which do not need
  to be located in contiguous memory cells
- Each node in a linked list can be stored in a different area of the computer's memory,
  instead of one large, contiguous block of memory
  - To do this, each node, consists of the data and a pointer to indicate the location
    of the next node in the linked list

- The start pointer/head pointer indicates the location of the first node in the linked list
- The last node in the linked list has a null pointer which does not point to any other node

▼ Differences between Linked List and arrays

	Array	Linked List
Size/Representation in memory	Arrays store elements in contiguous blocks of memory, with a fixed size determined at compile time.	Linked lists consist of individual nodes, each containing data and a reference to the next node. These nodes can be scattered across memory, allowing for dynamic resizing during runtime. Linked lists only use as much memory as they require as additional memory is allocated to the linked list only when it is required
Memory Efficiency	Arrays typically offer better memory efficiency, as they directly store data elements without additional overhead. However, they require contiguous memory allocation, which may lead to wastage of memory if the size exceeds the allocated space.	Linked lists may consume more memory due to the overhead of storing pointers to the next nodes along with data. However, when entries in the list need to be added, deleted or reordered, only the pointers need to be changed, which makes such operations much more efficient, especially when the data stored in each entry is large
Usage	Arrays are suitable for scenarios where the size is fixed and known beforehand. They offer efficient access to elements by index.	- Linked lists are preferred when flexibility in size is required, especially in cases of uncertainty or variation in data element sizes. They are well-suited for implementing dynamic

	Array	Linked List
		data structures such as stacks, queues, and graphs.
Execution Time	Arrays offer direct access to elements through their indices, resulting in faster access times for read and modification operations.	Linked lists may require traversal through multiple nodes to access or modify a specific element, which can lead to slower access times compared to arrays. However, they excel in insertion and deletion operations, as they can efficiently rearrange pointers without the need to shift contiguous memory blocks, making them more suitable for dynamic data manipulation.

▼ Time Complexity Linked List vs Array

Linked List:

Operation	Big-O	Note
Access	O(n)	Traversing the list sequentially
Search	O(n)	Traversing the list sequentially
Insert	O(1)	Assumes you have traversed to the insertion position
Remove	O(1)	Assumes you have traversed to the node to be removed

Array:

Operation	Big-O	Note
Access	O(1)	Direct access by index
Search	O(n)	Sequential search through unsorted array
Search (sorted array)	O(log(n))	Binary search
Insert	O(n)	Shifting subsequent elements
Insert (at the end)	O(1)	Special case: no shifting required

Operation	Big-O	Note
Remove	O(n)	Shifting subsequent elements
Remove (at the end)	O(1)	Special case: no shifting required