Definition

- These data structure are derived from primitive data structure
- More complex than primitive data structures
- Provides functionality to create custom data structures
- Emphesis on structuring a group of homogeneous or hetrogeneous data items

Types

- Files
- Lists
  - Linear List
    - Arrays
    - Stack
    - Linked List
      - Singly linked list
      - Doubly linked list
      - Circular linked list
    - Queues
      - Simple queue
      - Circular queue
      - Priority queue
      - Dequeue (Doubly Ended Queue)
  - Non-Linear List
    - Graphs
    - Trees

Linear List

- Data structure is linear if the data elements construct a sequence of a linear list
- The elements are adjacently attached to each other and in a specified

order
- The elments are linked one after other & represents linear relationship
- It consumes linear memory space & poor memory utilization

Array

- An array is sequential collcetion of homogenous data elements
- Stores the data element at contiguous(one after another) memory location.
- Data elements are accessed by using index
- Index always starts from 0

Time Complexity

Operation	**Complexity**
space	$O(n)$
lookup	$O(1)$
append	$O(1)$
insert	$O(n)$
delete	$O(n)$
search	$O(n)$ Sequential $O(log\ n)$ Binary search

Strength

- Fast lookups. Retrieving the element at a given index takes $O(1)$ time, regardless of the length of the array.
- Fast appends. Adding a new element at the end of the array takes $O(1)$ time.

Weakness

- Fixed size. You need to specify how many elements you're going to store in your array ahead of time. (Unless using a dynamic array.)
- Costly inserts and deletes. You have to "scoot over" the other elements

to fill in or close gaps, which takes worst-case $O(n)$ time.

Stack

- Ordered list of similar data elements
- Linear data structure, stores items in a **last-in, first-out (LIFO) order**.
- **push()** function is used to **insert** new elements into the Stack
- **pop()** function is used to **remove** an element from the stack.
- Inserstion and deletion takes place only at one end called ***top***
- **Overflow:** when stack is **completely full**
- **Underflow:** when stack if it is **completely empty**
- Element is inserted at top and only the element at top is removed making it as **LIFO**

Time Complexity

Operation	Complexity
space	$O(n)$
push	$O(1)$
pop	$O(1)$
peek	$O(1)$
search	$O(n)$

Top Stack Position

Position of Top	Status of Stack
-1	Stack is Empty
0	Only one element in Stack
N-1	Stack is Full
N	Overflow state of Stack

Basic Operations

Operation	Working
push()	Add element to top of stack
pop()	Remove element from top of stack
peek()	Get the top data element of the stack, without removing it.
isFull()	Check if stack is full.
isEmpty()	Check if stack is empty.

Implementation of stack

- Using Array
- Using Linked List

Application of stack

- In compilers : Expression evaluation
- Infix to Postfix /Prefix conversion
- In browsers : In keeping the previously visited URL's
- To revese a word / List
- Implementing function calls (recursion)
- Undo sequence in a text editor
- Other applications can be Backtracking, Knight tour problem, rat in a maze, N queen problem and sudoku solver

Linked List

- Data elements are linked to one another via a pointer
- Data elements are not necessarily stored in contiguous memory location
- Each element is called a node and has two parts:
    - Data
    - Reference to memory location of the next node
- The **first node** is called the **head**

- The **last node** is called the **tail**
- Also known as **self referential structure**

Time Complexity

Operation	**Complexity**
space	O(n)
prepend	O(1)
append	O(1)
lookup	O(n)
insert	O(n)
delete	O(n)

Operations on linked list

- **Insertion**
    - At the beginning
    - At the end
    - At specified location
- **Deletion**
    - From Beginning
    - From end
    - From specified position
- **Traversing**

Advantages

- Dynamic data structure - can grow and shrink druing execution time
- Easy insertion and deletion at desired position
- Insertion at end takes $O(1)$
- Deleting first element takes $O(1)$

Disadvantages

- More memory requied, if number of fileds are more.

- Costly lookups, access to arbitrary data item is time consuming

Applications of Linked List

- In implementation of stack, hash table, and binary tree
- Applications that have a Most Recently Used (MRU) lists
- Undo features in publishing or editing applications like Photoshop and Word.

Types of Linked list

- Singly Linked List - Item navigation is forward only.
- Doubly Linked List - Items can be navigated forward and backward.
- Circular Linked List - Last item contains link of the first element as next and the first element has a link to the last element as previous.

1. Singly Linked List

- Basic linked list
- Each node has only one address filed which points to next node
- Allows traversal in forward direction only

2. Doubly Linked List

- In a doubly linked list, each node have pointers to the next and the previous nodes.
- Allows traversal in both direction (forward and backward)

3. Circular Linked List

1. **Singly Linked List as Circular**
   The next pointer of the last node points to the first node

2. **Doubly Linked List as Circular**
   The next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.

> Linked List Key Points

> - **Null Pointer** - Address field of the last node contains NULL value to indicate the end of the list.
> - **External Pointer (Start Node)** - Contains pointer to the very first node of a linked list, enables to access entier linked list.
> - **Empty List** - If no nodes are persent, the list is empty list. at this time **Start = NULL**

Queue

- Non-Primitive Linear Data Structure
- Elements are **added** at one end called **REAR** or **tail**
- Elements are **removed/deleted** from another end called **FRONT** or **head**
- Queue follows FIFO (First-In-First-Out) Principle, data item stored first will be accessed first.
- Total number of Elements in Queue = (**FRONT - REAR + 1**)
- If **FRONT = REAR** Queue is empty
- Element added : **REAR = REAR + 1**
- Element removed : **FRONT = FRONT + 1**

Time Complexity

Operation	Time Complexity
space	O(n)
enqueue	O(1)
dequeue	O(1)
peek	O(1)

Basic Operation

Operation	Meaning
enqueue()	add (store) an item to the queue.
dequeue()	remove (access) an item from the queue.

Operation	Meaning
peek()	Gets the element at the front of the queue without removing it.
isfull()	Checks if the queue is full.
isempty()	Checks if the queue is empty.

Implementation of Queue

- Static implementation using Arrays
- Dynamic implementation using Linked List

Applications of Queue

- Breadth-first search uses a queue to keep track of the nodes to visit next.
- Printers use queues to manage jobs—jobs get printed in the order they're submitted.
- Web servers use queues to manage requests—page requests get fulfilled in the order they're received.
- Processes wait in the CPU scheduler's queue for their turn to run.

Types of Queue

- Simple queue
- Circular queue
- Priority queue
- Dequeue (Doubly Ended Queue)

1. Simple Queue

- The simple queue is a normal queue where insertion takes place at the **FRONT** of the queue
- and deletion takes place at the **END** of the queue.

2. Circular Queue

- Improvement over standard Queue Structure

- The last node is connected to the first node
- when an element is deleted, the vacant space is not reutilized.
- However, in a circular queue, vacant spaces are reutilized.
- While inserting elements, when you reach the end of an array and you need to insert another element, you must insert that element at the beginning
  (given that queue is full the first element has been deleted and the space is vacant).
- In add ition to standard variables one extra **Count** variable is use to keep track of No. of elements in Queue.

3. DEQUE (Double Ended Queue)

- Insertion and deletion can take place at both ends
- Because of the restriction to perform insertion or deletion only at one end,
  DEQUE has two types:
  - Input restricted DEQUE:
    - Deletion can be made from both ends, but insertion can be made at one end only.
  - Output restricted DEQUE:
    - Insertion can be made at both ends, but deletion can be made from one end only.

Basic Operations

Operation	**Meaning**
insertFront( )	Adds an item at the front of Deque.
insertLast( )	Adds an item at the rear of Deque.
deleteFront( )	Deletes an item from front of Deque.
deleteLast( )	Deletes an item from rear of Deque.
getFront( )	Gets the front item from queue.
getRear( )	Gets the last item from queue.

Operation	Meaning
isEmpty( )	Checks whether Deque is empty or not.
isFull( )	Checks whether Deque is full or not.

Implementation of DEQUE

- A Deque can be implemented either using a **doubly linked list** or **circular array**

Applications of Deque

- Deque supports both stack and queue operations, so it can be used as both.
- Supports clockwise and anticlockwise rotations in O(1) time
- Elements can be removed and or added both efficiently

4. Priority Queue

- Extension of standard queue
- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

Time Complexity

Operation	Time Complexity
space	$O(n)$
peek	$O(1)$
dequeue	$O(lg(n))$
enqueue	$O(lg(n))$

Advantages

- Quickly access the highest-priority item. Priority queues allow you to peek at the top item in $O(1)$ while keeping other operations relatively cheap $(O(lg(n))$.

Disadvantages

- Slow enqueues and dequeues. Both operations take $O(lg(n))$ time with priority queues. With normal first-in, first-out queues, these operations are $O(1)$ time.

Implementation of Priority Queue

- Using Array
- Using Heap *(preferred)*

Applications of Priority Queue

- Round Robin for processor scheduling
- Printer server routines are desined using queue
- Certain foundational algorithms rely on priority queues:
    - Dijkstra's shortest-path
    - A* search (a graph traversal algorithm like BFS)
    - Huffman codes (an encoding for data compression)
- All queue applications where priority is involved.