Data Structure

Data structures are used by computers as the containers within which information is stored. Different data structures exist and some are better suited to different types of data than others. When storing data, a programmer must decide which of the data structures available is the best to use.

What is a data Structure?

Data Structure is a way of collecting and organizing data in the computer in such a way that different operations such as searching and retrieving, on these data can be done in an effective manner.

Anything that can store data in the computer can be called a data structure.

They are containers used by the computer to store data

Types of Data Structure

There are different data structure that is used by the computer to store data

These are divided in two types

* Primitive
* Non-primitive

Primitive Data Structure

Diagram

Description automatically generatedPrimitive Data Structure is also called Built-in Data structures because they are built into a programming language. They are atomic and cannot be divided further

They are the basic data types that are available in most of the programming languages.

Examples of primitive data structure

Integer, Float, Boolean, Char etc

Diagram

Description automatically generated

Non-Primitive Data Structures

**Also called User Defined Data Structure**

They are complex Structures, that are used to store large groups of connected data.

They are not atomic and **cannot be formed without using** primitive data structure

**Abstract data Types (ADT)**

Data type that is defined by its behaviour and operation rather than its implementation. It describes a set of data value and the operations that can be performed on those value, without specifying how these values are actually stored or manipulated in memory.

**Two types**

* Stacks
* Queues
* Linked lists

**Data Structures: Stacks & Queues**

Chart, diagram, box and whisker chart

Description automatically generatedStacks and Queues are both commonly used data structures that allow us to dynamically store and retrieve data items in two very different ways. We can use these in a variety of situations; however, the choice depends on the problem that we are trying to solve.

### **STACK**

Diagram, schematic, waterfall chart

Description automatically generatedA stack is a dynamic linear data structure in which all the insertion and deletion of data/ values are done at one end only, rather than in the middle. It uses the **Last-In-First-Out** principle (**LIFO**) to define data structure - this structure limits data in the way that it can only be added to or removed from the top.

The delete operation is known as **POP**, in relation to the physical idea of popping something from a stack - the **LIFO** structure also means we can only remove the item that was most recently added. In order to add items to a stack, the **PUSH** operation can be used, which refers to the physical idea of pushing an item onto the top of a stack.

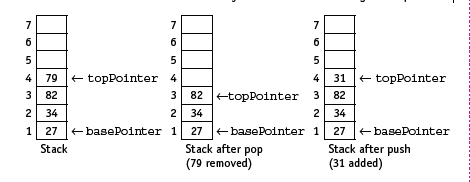
As well as the **PUSH** and **POP** operations we also have access to:

* **peek** — Used to return the item at the top of the stack
* **isEmpty** — Used to check whether the stack contains items

A stack uses two pointers:

* a base pointer points to the first item in the stack
* a top pointer points to the last item in the stack.

When they are equal there is only one item in the stack.

The value of the basePointer always remains the same during stack operations:

Operating of a Stack

Diagram

Description automatically generated

The first Item is pushed to the bottom of the stack and the next item is added on top of the first ect

Taking an Item off the stack is called popping. The item that is added last is the one that taken off first

There are many situations where a stack can be useful, a few implementations of it’s use could be found in:

* **Undo/backtracking:** Stack is also used to undo in text-editor. The stack simply allows us to pop the previous item from it’s data structure.
* **Forward and backward** feature in web browsers
* **Expression conversion**: An expression is represented in the prefix, postfix, or infix notation. The stack is used to convert the form of one expression to another form.
* **Syntax parsing**: Many compilers use the stack to parsing the syntax expressions.
* **Memory management**: It is also used in memory management.
* **Variable tracking:** Stack is also used to track local variables in runtime.
* **Parenthesis checking:** Stack is also used to check whether parenthesis is correctly open and closed or not.
* **String reverse:** Stack is also used to reverse the string. In the string, we push the characters into the stack one by one and then pop the characters from the stack one by one.

Using an array implementation for a stack is a simple approach and involves keeping reference to just a few indexes. The top index keeps a reference of the top item in our stack, whilst the capacity index keeps us informed about the size of our array. We can use these together to be aware of how full/empty our stack is at the location of the next item to be removed. If we were using a dynamic implementation then when our **top** index reaches the **capacity** index, we simply double the size of the stack.

### Queue

QUEUE is another extremely common type of list data structure.

Chart, waterfall chart

Description automatically generated

It is implemented on a **First-In-First-Out** (**FIFO**) principle, meaning that the items are removed from the list in exactly the same order in which they were added to it.

For example, it simulates the physical approach of a queuing systems used whilst waiting in line at your favourite food store - the first in line will be the first to leave, with new people being, added to the back of the queue.

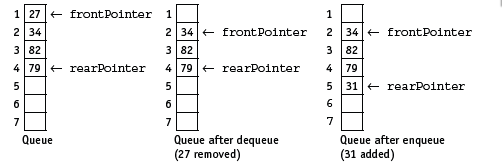
When we insert an item onto the queue this is known as the **ENQUEUE** function, whilst deleting an item is referred to as the **DEQUEUE** function.

Table

Description automatically generated with medium confidenceIt maintains two pointers

* A front pointer
* An 'rear' pointer

The value of the **frontPointer** changes after **dequeue** but the value of the **rearPointer** changes after **enqueue**:



Items are removed from the front and added to the end of a queue, the position of the queue in the array changes. Therefore, the queue should be managed as a circular queue to avoid moving the position of the items in the array every time an item is removed.

When a queue is implemented using an array with a finite number of elements, it is managed as a circular queue. Both pointers, frontPointer and rearPointer, are updated to point to the first element in the array (lower bound) after an operation where that pointer was originally pointing to the last element of the array (upper bound), providing the length of the queue does not exceed the size of the array.

### Most times queues are used whenever there are a number of items waiting for a resource to become available.

## Applications of queue:

1. Interrupts handling: A queue can be used to store a list of interrupts in the operating system, which would get processed in the order in which they were generated.
2. Time sharing OS. A queue can be found in a time-sharing computer system where many users share the system simultaneously and share the space of the memory.
3. Computer Network (LAN). Messages send are kept in a buffer until the receiving computer is ready for it. Messages also need to be sent to receiving computer in the same order in which they are created, i.e. FIFO order.
4. Used in printing. The jobs are printed in the order they are received. (FIFO)

**Types of Queues**

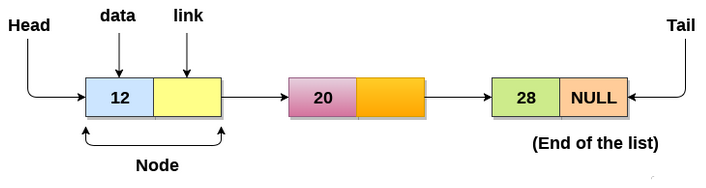
* Circular queue This is a particular kind of queue where new items are added to the rear of the queue as items are read off the front of the queue, So there is constant stream of data flowing into and out of the queue. Another name for it is 'circular buffer. This kind of queue reduces wastage of space in case of array implementation
* **Linear Queue** works its way through the allotted memory without reusing the memory.
* Priority Queue: An element is given a value to indicate its priority. Items with the highest is process first. 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

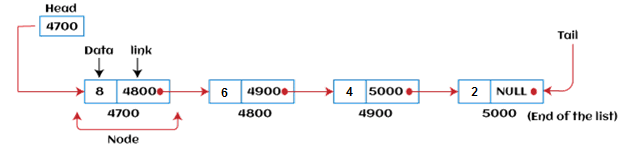
List

When data items (also called elements or nodes) are related to one another, they can be organised as a list. In a list there is a pattern of relationship that is universal therefore a list is the most basic data structure from which flows much more complicated structures (array, stack, queue, tree, graph) - but in the end, they are all just a type of list.

Linked list is a linear data structure that includes a series of connected nodes. Linked list can be defined as the nodes that are randomly stored in the memory. A linked list is a dynamic data structure, which means that the size of the list can change at run time.

A node in a linked list contains two fields

* Data part: Data stored at that particular address
* Address part: The pointer which contains the address of the next node in the memory.
* Each node in the list is stored in memory, one after the other
* Every link contains a connection to another link.
* The last node of the list contains pointer to the null.
* The first node of the list is called the head, and the last node is the tail.

The following is a linked list with four nodes (each containing a number)

There is also a separate pointer that indicates the first element in the list (the head of the list). This has a null value when the list is empty. An additional pointer can be used to indicate the last element of the list (the **tail** of the list). This is useful when implementing a queue using a linked list so that the end of the list can be accessed without needing to traverse the list.

Types of Linked list

Linked list is classified into the following types -

* Singly-linked list - Singly linked list can be defined as the collection of an ordered set of elements. A node in the singly linked list consists of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node, while the link part of the node stores the address of its immediate successor.
* Doubly linked list - Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly-linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer), and pointer to the previous node (previous pointer).
* Circular singly linked list - In a circular singly linked list, the last node of the list contains a pointer to the first node of the list. We can have circular singly linked list as well as circular doubly linked list.
* Circular doubly linked list - Circular doubly linked list is a more complex type of data structure in which a node contains pointers to its previous node as well as the next node. Circular doubly linked list doesn't contain NULL in any of the nodes. The last node of the list contains the address of the first node of the list. The first node of the list also contains the address of the last node in its previous pointer.

Operations performed on Linked list

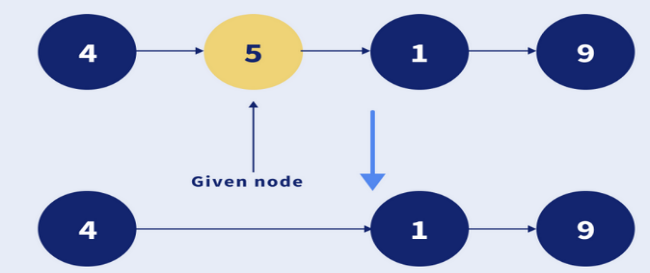
The basic operations that are supported by a list are mentioned as follows -

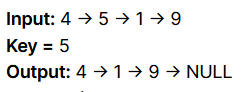
* Insertion - This operation is performed to add an element into the list.
* Deletion - It is performed to delete an operation from the list.
* Display - It is performed to display the elements of the list.
* Search - It is performed to search an element from the list using the given key.

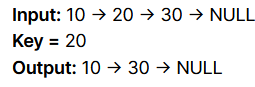
Deletion Operation

**Problem Statement**

Given the linked list. The task is to delete the node of the linked list and return the list.

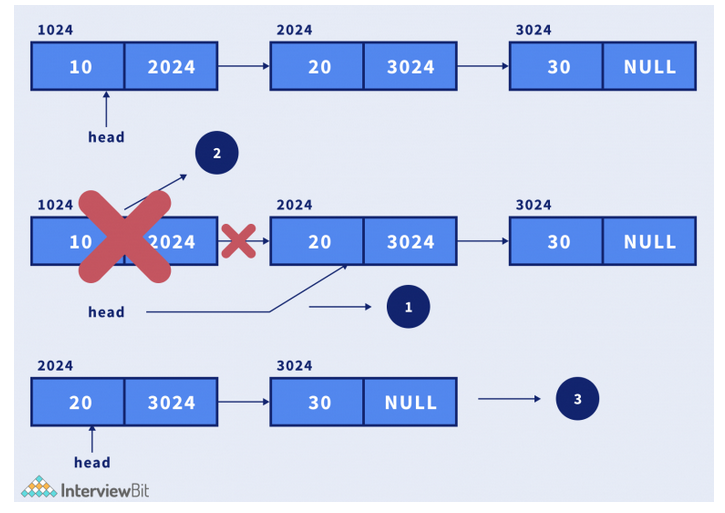
**Examples:**



Write the return list for the following

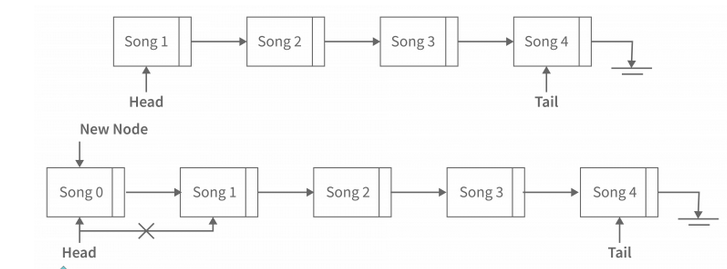
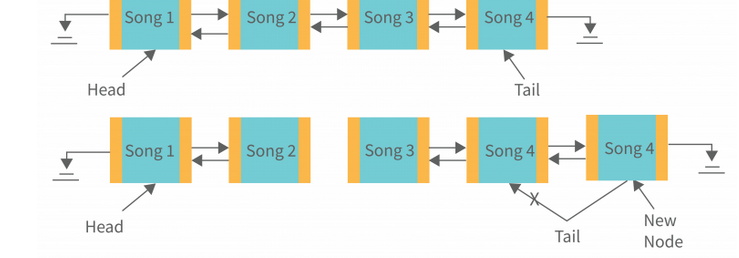
**Algorithm**

* If the node to be deleted is the head node, then simply point the head to the second node of the linked list.
* For all other nodes:
  + Traverse the linked list and for the current node **curr,** check whether the next node contains the key that needs to be deleted.
    - If true, point **curr -> next** to **curr -> next -> next**.
    - Else, keep traversing the list until the end of the linked list.

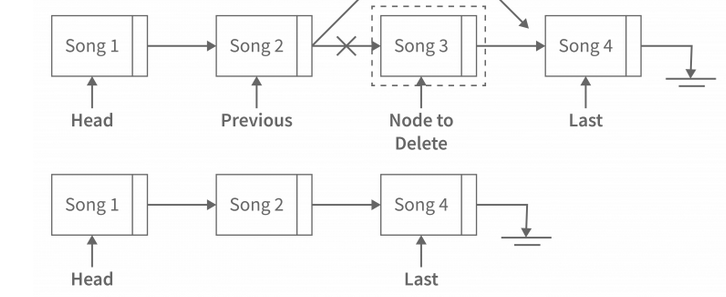


Inserting a node

1. Create a node using the defined structure.
2. Assign the data to the data part of the node.
3. For the Next pointer,
   1. **Inserting at front –** Assign the next pointer to the head node and make the head node point to this new node.
   2. **Inserting at the end –** Assign the tail node next pointer point to the new node and make the tail point to the new node and then make the new pointer next node point to null to represent the end of the list.
   3. **Inserting in the middle –** Search the position to be inserted and adjust the pointer of new node points to the next location of the position. For the positioning node, make its next pointer point to the new node.



Deletion



Algorithm for Deletion from Singly Linked-List –

* Search for the node to be deleted.
* If the node wasn’t found then return a message indicating the node was not found.
* If node found –
* Along with searching, use an extra tail pointer that will point to the previous node of the current while searching.
* Adjust the next pointer of that tail pointer to the node to delete the next pointer. And free that node to delete memory.

#### Uses of Linked List

#### The list is not required to be contiguously present in the memory. The node can reside any where in the memory and linked together to make a list. This achieves optimized utilization of space.

#### List size is limited to the memory size and doesn't need to be declared in advance.

#### Empty node cannot be present in the linked list.

#### We can store values of primitive types or objects in the singly linked list.

#### Why use linked list over array?

#### Till now, we were using array data structure to organize the group of elements that are to be stored individually in the memory. However, Array has several advantages and disadvantages which must be known in order to decide the data structure which will be used throughout the program.

#### Array contains following limitations:

#### The size of array must be known in advance before using it in the program.

#### Increasing size of the array is a time taking process. It is almost impossible to expand the size of the array at run time.

#### All the elements in the array need to be contiguously stored in the memory. Inserting any element in the array needs shifting of all its predecessors.

#### Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because,

#### It allocates the memory dynamically. All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.

#### Sizing is no longer a problem since we do not need to define its size at the time of declaration. List grows as per the program's demand and limited to the available memory space.