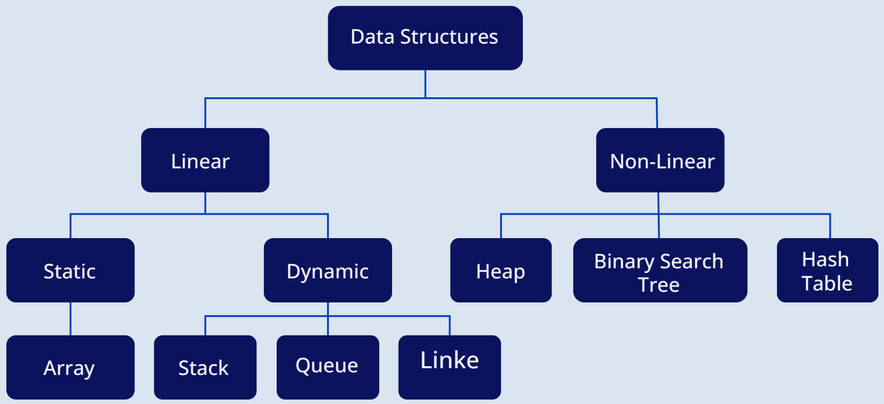
7 ESSENTIAL DATA STRUCTURES FOR CODING INTERVIEWS

1. Array
2. Linked List
3. Hash Table
4. Stack
5. Queue
6. Heap
7. Binary Search Tree

What is a Data Structure?

A data structure organizes collections of data to store it efficiently to make the process of accessing or modifying the data easier.

Data Structure Categories



Time and Space Complexity

- **Time complexity**: The amount of time required to run analgorithm, corresponding to an input of a specific size not including the space taken by the input.

  Space complexity: The amount of space required to run an algorithm, corresponding to an input of a specific size. not including the space taken by the input.

Different types of complexities

Complexity	Performance	Type
O(1)	Best	Constant
O(log n)	Excellent	Logrithmic
O(n)	Good	Linear (Polynomial)
O(n log n)	Fair	Polynomial
O(n2)	Bad	Quadratic (Polynomial)
O(2n)	Poor	Exponential
O(n!)	Worst	Factorial

Basic operation on data structures

Operations	Description
Access	Visits an element in a data
Search	Searches for an element in the data structure
Insert	Inserts an element in the data
Delete	Deletes an element from the data structure

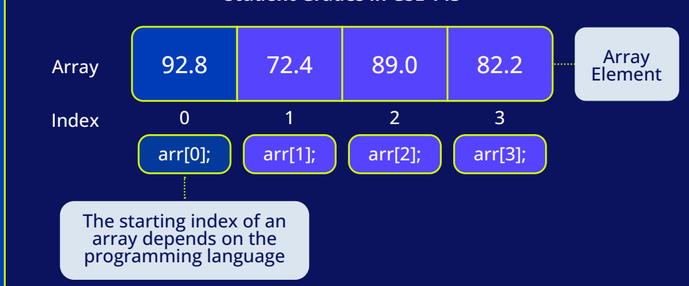
1. Array

An array is a linear data structure that contains elements of the same data type stored at contiguous memory locations.

- **Element:** A data item stored in an array at a specific index
- **Index:** A unique, numerical identifier that indicates an element's location in an array

Pros	Cons
○ Great for storing and accessing large amounts of data quickly, as they provide constant time access to all elements, regardless of their index position ○ Good for storing multiple elements of the same type ○ Easy to sort and search through	○ Arrays are fixed-size data structures, so expanding or minimizing one requires reallocation. This can consume a lot of time and memory.

Array example



Array Operations	Time Complexity (Average)	Time Complexity (Worst Case Scenario)	Space Complexity (Worst Case Scenario)
Search	$\theta(n)$	$O(n)$	$O(n)$
Insert	$\theta(n)$	$O(n)$	
Delete	$\theta(n)$	$O(n)$	

Code example

```cpp
// C++ code to print an array:
#include <iostream>
using namespace std;
// This function prints an array
void printArray(int *arr,int size)
{
    cout<<"The contents of the array are:\n";
    for(int i=0; i<size;i++)
       cout<<arr[i]<<" ";
    cout<<endl;
}
// Driver Code
int main()
{
       // Considering inputs given are valid
       int arr[5]={10,20,30,40,50};
       printArray(arr,5);
       return 0;
}
```

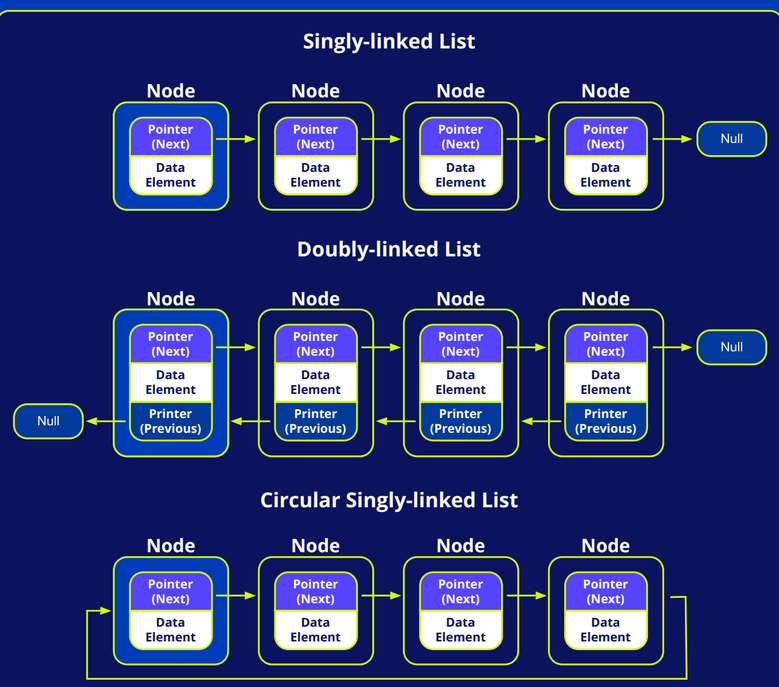
Code to print an array in C++

Pros	Cons
○ Dynamic data structure: adding and deleting nodes is easier ○ More efficient for data of arbitrary length whose size is not known in advance ○ Can store data even when no contiguous memory is available	○ Retrieving nodes takes more time as it doesn't support direct access to individual elements via index positions in constant time ○ Searching data is slow as techniques such as binary search cannot be employed ○ Nodes with pointers use up some memory

Types of linked lists

- **Singly-linked list:** Only allows forward navigation
- **Doubly-linked list:** Allows forward and backward navigation Unlike a typical linked list, a doubly-linked list maintains two
- C**ircular linked list:** The node in this list has the next pointer to the first node

Linked list example



Linked List * Operations	Time Complexity (Average)	Time Complexity (Worst Case Scenario)	Space Complexity (Worst Case Scenario)
Search	$\theta(n)$	$O(n)$	$O(n)$
Insert	$\theta(1)$	$O(1)$	
Delete	$\theta(n)$	$O(n)$	

Singly-Linked **and** Doubly-Linked

2. Linked list

A linked list is a linear data structure that stores data as a chain of nodes at non-contiguous memory locations linked together by pointers. Each node contains a data element and pointer to the

- **Data:** Heterogeneous data can be stored in each node
- **Pointer:** It is used to store the address of the next node

Code Example

```cpp
#include <iostream>
using namespace std;

// Making a node struct containing an int data and a pointer
// to next node
struct Node
{
    int data;
    Node *next;
    // Parameterised constructor with default argument
    Node(int val=0) :data(val),next(nullptr){}
    // Parameterise constructor
    Node(int val, Node *tempNext):data(val),next(tempNext){}
};

class LinkedList
{
    // Head pointer
    Node* head;

 public:
    // default constructor. Initializing head pointer
    LinkedList():head(nullptr)
    {
    }

    // inserting elements (At start of the list)
    void insert(int val)
    {
    // make a new node
    Node *new_node = new Node(val);
    I// If list is empty, make the new node, the head
    if (head == nullptr)
    {
      head = new_node;
    }
    // else, make the new_node the head and its next, the previous
    // head
    else
    {
      new_node->next = head;
      head = new_node;
    }
  }


    void display()
    {
        Node* temp = head;
        while(temp != nullptr)
        {
            cout << temp->data << " ";
            temp = temp->next;
        }
        cout << endl;
    }
};
int main()
{
    LinkedList l;
    // inserting elements
    l.insert(6);
    l.insert(9);
    l.insert(1);
    l.insert(3);
    l.insert(7);
    cout << "Current Linked List: ";
    l.display();
}
```

Linked list class in C++

```python
# A single node of a singly linked list
class Node:
    # constructor
    def __init__(self, data =None, next=None):
        self.data = data
        self.next = next

# A Linked List class with a single head node
class LinkedList:
  def __init__(self):
        self.head = None

    # insertion method for the linked list
    def  insert(self, data):
        newNode = Node(data)
        if self.head:
            current = self.head
            while current.next:
                current = current.next
            current.next = newNode
        else:
            self.head = newNode

    # print method for the linked list
    def printLL(self):
        current = self.head
        while(current):
            print(current.data)
            current = current.next

# Singly Linked List with insertion and print methods
LL = LinkedList()
LL.insert(6)
LL.insert(9)
LL.insert(1)
LL.insert(3)
LL.insert(7)
LL.printLL()
```

Linked list class in Python

3. Hash table

A **hash table** is an **unordered** collection of **key-value pairs** where a **hash key** is mapped to a **hash value** (data element), typically as an array of linked lists. A hash function takes arbitrary-length data as an input and generates a fixed-length output. The output is the **hash key** of the given data. A value is stored at a location based on its hash key. This process is known as **hashing**.

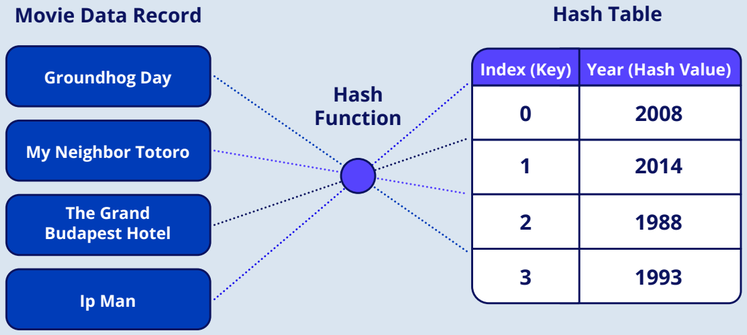
In summary

Hash function: Takes an arbitrary length input and produces a fixed-length output
Hash value: Input of a hash function
Hash key: Output of a hash function

If the hash function generates the same hash keys for multiple hash values, then a conflict known as a hash collision occurs, where two pieces of data in a hash table share the same hash value.

Pros	Cons
○ Constant time lookup, insertion and deletion (as long as there is no hash collision) ○ May store heterogeneous data	○ Complex to implement ○ Hash collisions can cause performance issues when accessing or searching items in a hash table ○ Hash keys require some extra computational

Hash table example



Hash Table Operations	Time Complexity (Average)	Time Complexity (Worst Case Scenario)	Space Complexity (Worst Case Scenario)
Search	$\theta(1)$	$O(n)$	$O(n)$
Insert	$\theta(1)$	$O(n)$	
Delete	$\theta(1)$	$O(n)$	

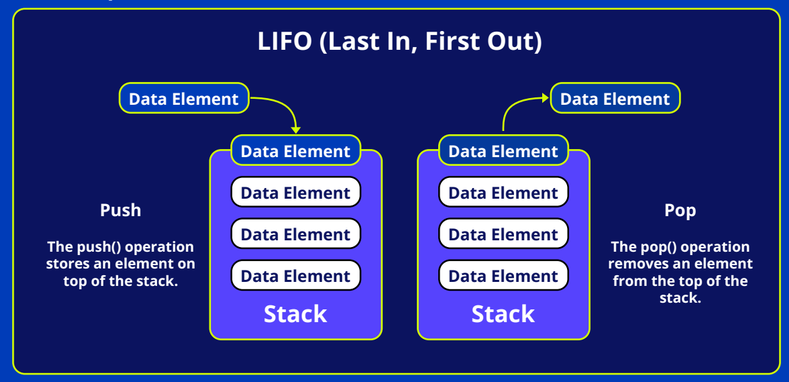
4. Stack

Stack and **queue** data structures are often mentioned in the same breath because of their similar characteristics, but they behave differently when adding or removing data.

Stacks store elements in a Last-In, First-Out (LIFO) order, meaning that the last element added to the stack is the first one removed.

Pros	Cons
○ Great for depth-first search (DFS) ○ Efficient addition/removal of items ○ Used in recursive function calls and also by the call-stack of programming languages	○ Limited use cases compared to some other data structures

Stack example



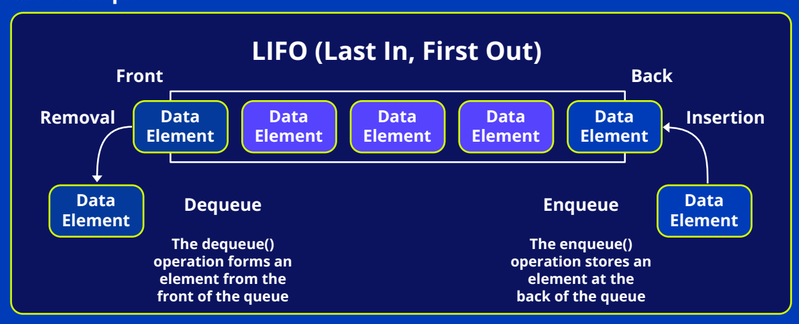
Stack Operations	Time Complexity (Average)	Time Complexity (Worst Case Scenario)	Space Complexity (Worst Case Scenario)
Search	$\theta(n)$	$\theta(n)$	$\theta(n)$
Insert	$\theta(1)$	$\theta(1)$	
Delete	$\theta(1)$	$\theta(1)$	

5. Queue

Queues store elements in a First-In, First-Out (FIFO) order, meaning that elements are removed in the order they arrived.

Pros	Cons
○ Great for breadth-first search (BFS) ○ Efficient add/removal of items	○ Limited use cases compared to other data

Queue example



Queue Operations	Time Complexity (Average)	Time Complexity (Worst Case Scenario)	Space Complexity (Worst Case Scenario)
Search	$\theta(n)$	$O(n)$	$O(n)$
Insert	$\theta(1)$	$O(1)$	
Delete	$\theta(1)$	$O(1)$	

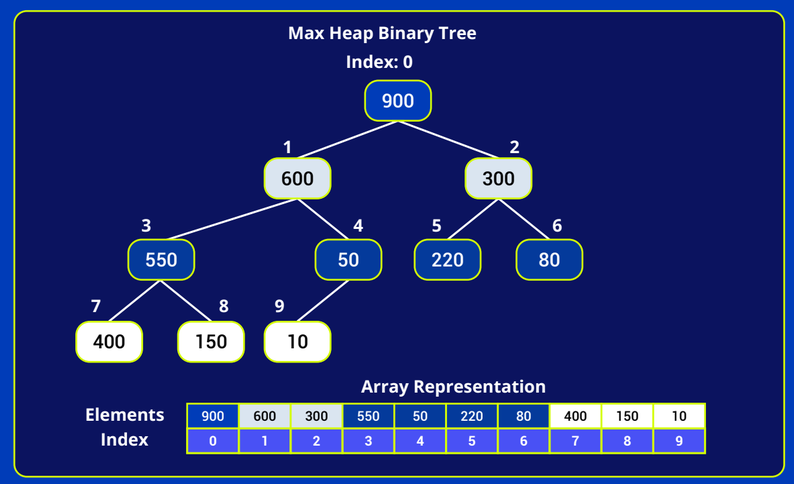
6. Heap

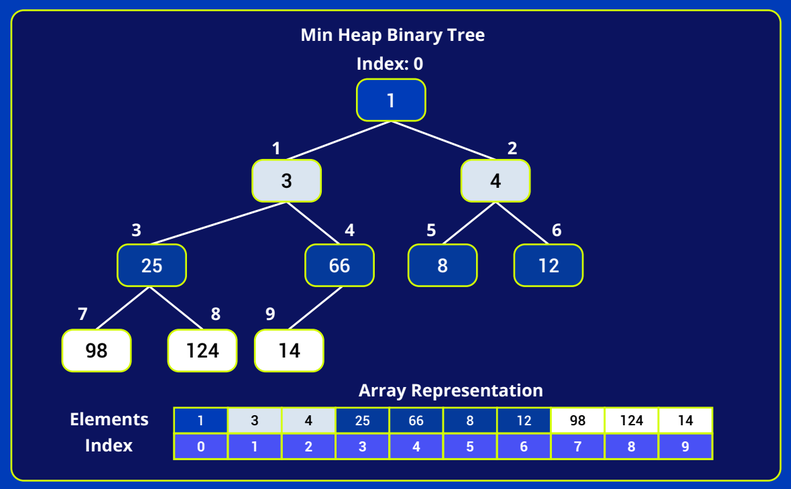
A **heap** is a complete binary tree where each node is associated with a key that satisfies a **heap property**. Heap data structures are particularly useful for finding the minimum or maximum value in a data set, and are effectively used as priority queues.

Max Heap Property: In the case of a **max heap**, the keyof a parent node must be **greater than or equal to** the keyof the child node.

Min Heap Property: In the case of a **min heap**, the keyof a parent node must be **less than or equal to** the keyof the child node.

Pros	Cons
○ Time complexity of insertion or deletion is just O(log n) ○ A binary heap is a complete binary tree ○ Simple to implement, can be done with just an array	○ Not the best data structure to traverse or search data





Heap Operations	Time Complexity (Average)	Time Complexity (Worst Case Scenario)	Space Complexity (Worst Case Scenario)
Search	$\theta(n)$	$O(n)$	$O(\log n)$
Insert	$\theta(\log n)$	$O(\log n)$	
Delete	$\theta(\log n)$	$O(\log n)$	

Application of Heap Data Structures

- Priority queues
- Heapsort
- Order statistics

7. Binary Search Trees

Binary search trees (BST) are non-linear data structures that store data elements in nodes. Each node contains one or more data items and two pointers that branch off to child nodes called the **left child** and the **right child**.

- The key of the left child must be less than or equal to the key of the parent node.
- The key of the right child must be greater than the key of the parent node.

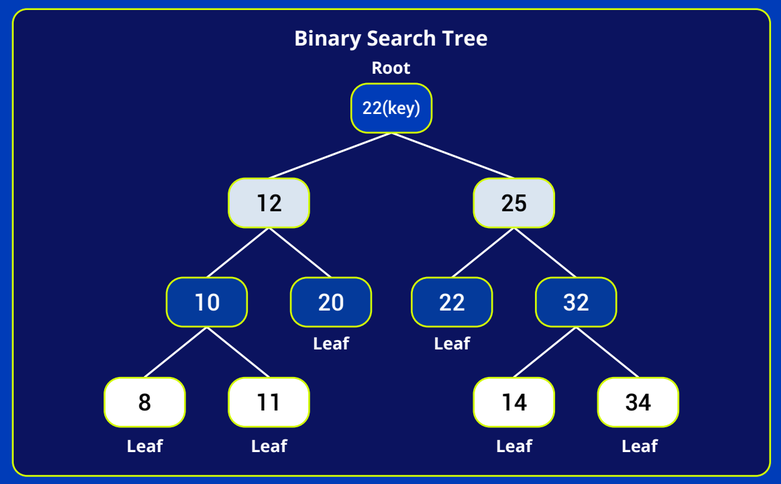
This allows you to use relational operators to efficiently find items in a binary search tree.

That is, we compare a key (x) stored in a node x with the value to search y, using the following three conditions:

1. When x is equal to y, x==y, end the search
2. When x is less than or equal to y, x<=y, go to the right child
3. When x is greater than y, x>y, go to the left child

Pros	Cons
○ Insertion and deletion can be fast and efficient ○ Supports range queries - Relatively simple compared to other data structures ○ Easy to implement	○ Implementing a balanced binary search tree is necessary to avoid degradation of performance ○ Accessing elements from a BST is slower than in an array

Binary search tree example



Binary Search Tree Operations	Time Complexity (Average)	Time Complexity (Worst Case Scenario)	Space Complexity (Worst Case Scenario)
Search	$\theta(\log n)$	$O(n)$	$O(n)$
Insert	$\theta(\log n)$	$O(n)$	
Delete	$\theta(\log n)$	$O(n)$	

A key issue with BST is that they are not self-balancing hence leading to compromised time-complexity. To mitigate this, several self-balancing BST variants can be used, such as **AVL Trees** and **Red Black Trees**.

Advanced Data Structures

We've covered 7 essential data structures that every developer should learn.

Some advanced data structures that software engineers can get familiar with include:

1. Tries
2. Self-Balancing BSTs
3. Disjoint Sets
4. Segment Trees
5. Queaps
6. Binary Indexed Trees
7. Succinct Data Structures