Made by CaiNiao with ❤️

Algorithms & Connectionist AI

STUDY GUIDE

ALGORITHMS & CONNECTIONIST AI

Introduction

This Study Guide, prepared by CaiNiao, will walk you through three main topics:

Algorithms: Written by Anmol, this section will introduce the core principles of algorithms, including sorting, searching, and optimization, equipping you with the tools to solve problems efficiently.

Data Structures: Written by Revell, here you'll learn about the fundamental data structures that support algorithmic performance, focusing on arrays, lists, trees, graphs, and hash tables.

Connectionist AI: Written by Russel, this section delves into neural networks and connectionist AI, exploring how machines learn and adapt from data, and the impact of this on fields like computer vision and language processing.

This study guide is designed to help you understand two key areas of computer science: the structured problem-solving methods of algorithms, and the adaptive, data-driven nature of connectionist AI. By exploring both, you'll gain a strong foundation in how computers solve problems and learn from data, which are crucial skills in today's tech-driven world. You'll be tested on these concepts through interactive learning activities, so it's essential to fully engage with the material and become familiar with it.

Table of Contents

INTRODUCTION TO ALGORITHMS....... **1**
Introduction....... 1
Learning Suggestions:....... 2
Running Time....... 5
SEARCHING ALGORITHMS....... **7**
Searching Algorithms....... 8
Linear Search....... 8
Binary Search....... 14
BFS(Breadth-First Search)....... 16
DFS(Depth First Search)....... 18
SORTING ALGORITHMS....... **21**
Sorting Algorithms (Iterative/introductory sorts)....... 21
Selection sort....... 22
Bubble Sort....... 24
Insertion Sort....... 25
Recursion....... 27
Sorting Algorithms (Recursive/divide and conquer sorts)....... 32
Quick sort....... 35
INTRODUCTION TO DATA STRUCTURES....... **39**
Introduction....... 40
The importance of data structures....... 40

Learning Suggestions....41
LINEAR DATA STRUCTURES....42
Linear Data Structures.... 43
Static Data Structure.... 43
Dynamic Data Structure.... 48
NON-LINEAR DATA STRUCTURES.... 59
Non-linear Data Structures.... 60
Trees.... 61
Graphs.... 69
HASH-BASED DATA STRUCTURES....74
Hash-Based Data Structures.... 75
Hash Table.... 76
Conclusion.... 82
INTRODUCTION TO CONNECTIONIST AI....84
Introduction.... 85
Key Characteristics....85
Learning Suggestions....86
HOW DOES AN AI LEARN?....87
How Does an AI Learn?....88
1. Supervised Learning.... 88
2. Unsupervised Learning....90
3. Reinforcement Learning.... 92
Credit Assignment.... 93
NEURAL NETWORKS....96
Neural Networks....97
Structure of ANN.... 97
POPULAR NEURAL NETWORKS....105
Popular Neural Networks....106
Feedforward Neural Network (FNN).... 106

Recurrent Neural Network (RNN) ... 108
Long Short-Term Memory (LSTM) ... 111
Gated Recurrent Unit (GRU) ... 114
Convolutional Neural Network (CNN) ... 118
Radial Basis Function Network (RBFN) ... 122
KEY APPLICATIONS OF CONNECTIONIST AI ... 126
Key Applications of Connectionist AI ... 127
Image Recognition ... 127
Speech Recognition ... 127
Autonomous Driving ... 128
Medical Diagnosis ... 129
Time Series Prediction ... 129
Robotics ... 129
Recommendation Systems ... 130
CHALLENGES IN CONNECTIONIST AI ... 131
Challenges in Connectionist AI ... 132
Data Requirements ... 132
Interpretability ... 132
Overfitting ... 133
Bias and Fairness ... 134
Generalization ... 134
How to mitigate these challenges in the future ... 135
Data Requirements ... 135
Interpretability ... 136
Overfitting ... 136
Bias and Fairness ... 137
Generalization ... 137
ETHICAL CONSIDERATIONS ... 139
Ethical Considerations ... 140

Bias and Fairness ... 140
Transparency and Explainability ... 141
Privacy and Data Security ... 141
Employment and Societal Impact ... 141
Misuse and Security Risks ... 142

Mastering Algorithms

A practical playbook
VOL.1

2025

Algorithmic Ventures

Presented by
Cainiao

INTRODUCTION TO ALGORITHMS

In this chapter:

- Introduction to algorithmic thinking
- Running time
- Learning suggestions

Introduction

Problem-solving is central to computer science and computer programming. An algorithm is a step-by-step set of instructions to solve a problem.

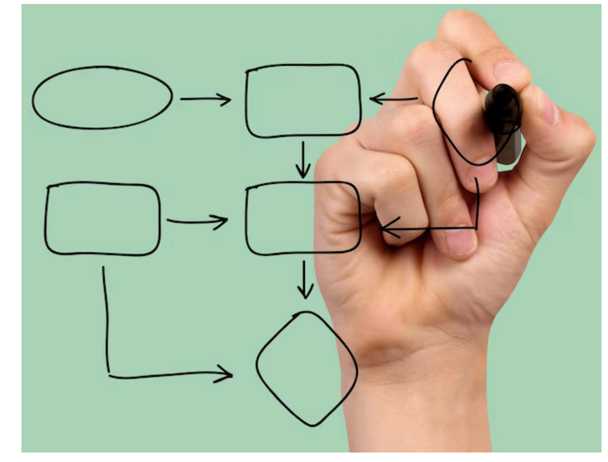
Imagine the basic problem of trying to locate a single name in a phone book.

How might one go about this? One approach could be to simply read from page one to the next until reaching the last page. Another approach could be to search two pages at a time. A final and perhaps better approach could be to go to the middle of the phone book and ask, “Is the name I am looking for to the left or to the right?” Then, repeat this process, cutting the problem in half and half and half.

Each of these approaches could be called algorithms. The speed of each of these algorithms can be pictured as follows in what is called big-O notation:

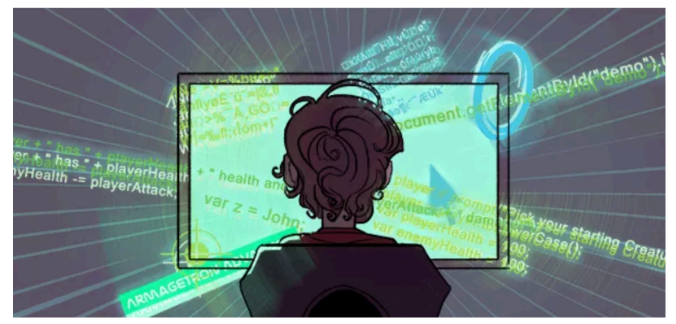
Learning Suggestions:

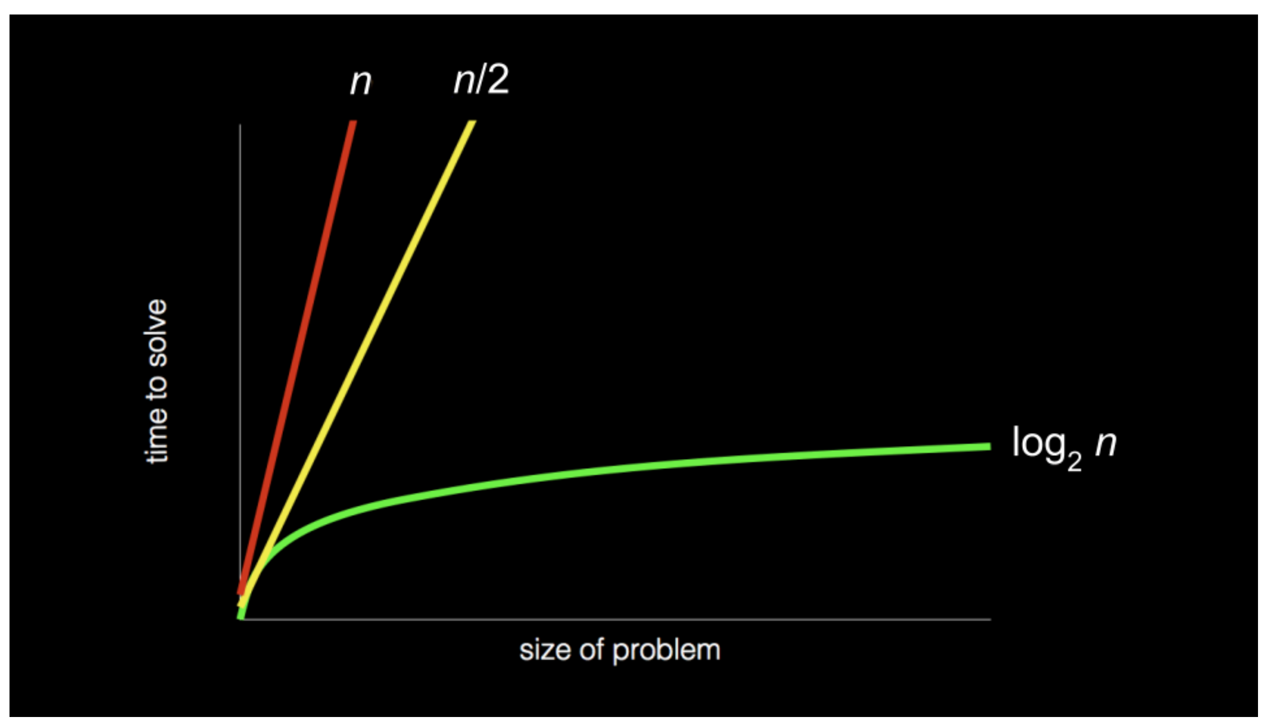
1. **Draw the sorting/searching algorithms on paper:** This can be especially helpful for visual learners. You can take inspiration from Googling GIFs of specific sorting algorithms. Having a visual understanding of how the searching/sorting algorithms work can make learning faster!



2. **Code it:** Although programming languages (notably high-level ones) may already have built-in functions to sort/search efficiently, and you

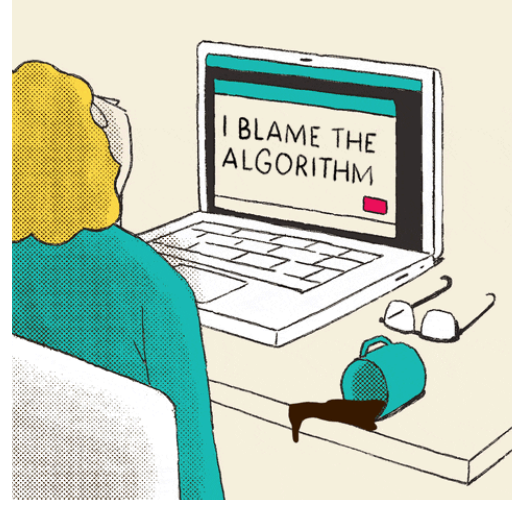
may ask yourself in this case, “Then what’s the point of coding it myself?” It is still great to not only practice coding, but also to understand how the computer runs the algorithm. For beginners, Python is recommended.





We are going to consider the efficiency of these algorithms. Indeed, we are going to be building upon our understanding of how to use some of the concepts we will know from these algorithms. You should consider how the way an algorithm works with a problem may determine the time it takes to

solve a problem! Algorithms can be designed to be more and more efficient to a limit.

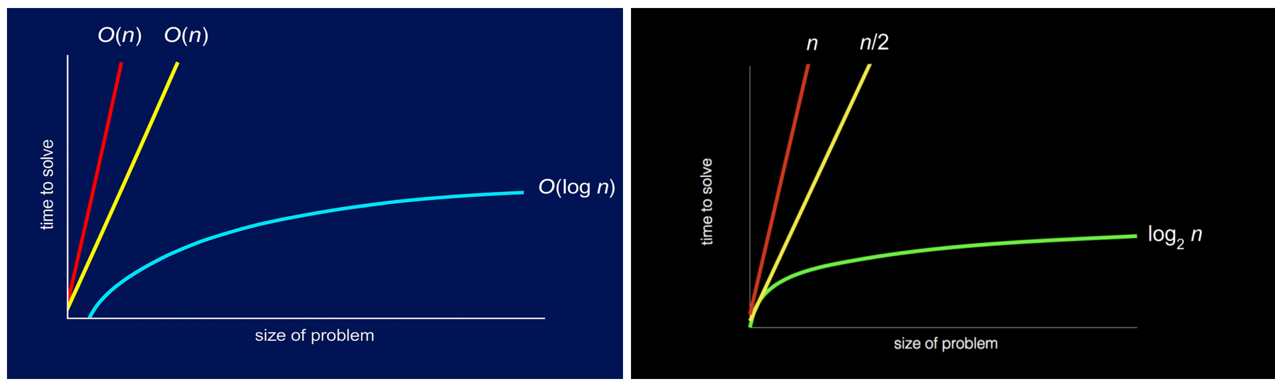


Notice that the first algorithm, highlighted in red, has a big-O of n because if there are 100 names in the phone book, it could take up to 100 tries to find the correct name if we go one by one through every page. The second algorithm, where two pages were searched at a time, has a big-O of n/2 because we searched twice as fast through the pages. The final algorithm has a big-O of log2n, as the final algorithm divides the problem in half every single time, so doubling the problem would only result in one more step to solve the problem.

- We are going to expand upon our understanding of algorithms through pseudocode and into code itself.
- We will focus on the design of algorithms and how to measure their efficiency.

RUNNING TIME

- You can consider how much time it takes an algorithm to solve a problem.
- Running time involves an analysis using big O notation. Take a look at the following graph:



Rather than being ultra-specific about the mathematical efficiency of an algorithm, computer scientists discuss efficiency in terms of the order of various running times.

In the above graph, the first algorithm is $O(n)$ or in the order of n. The second is in $O(n)$ as well. The third is in $O(\log n)$.

It's the shape of the curve that shows the efficiency of an algorithm. Some common running times we may see are:

$O(n2)$, $O(n\log n)$, $O(n)$, $O(\log n)$, $O(1)$.

Of the running times above, $O(n2)$ is considered the slowest running time. $O(1)$ is the fastest. Linear search was of order $O(n)$ because it could take n steps in the worst-case to run. Binary search was of order $O(\log n)$ because it would take fewer and fewer steps to run, even in the worst-case.

Programmers are interested in both the worst-case, or upper bound, and the best-case, or lower bound. The Ω symbol is used to denote the best-case of an algorithm, such as $\Omega(\log n)$. The Θ symbol is used to denote where the upper bound and lower bound are the same: Where the best-case and the worst-case running times are the same. Asymptotic notation is the measure of how well algorithms perform as the input gets larger and larger. As you continue to develop your knowledge in computer science, you will explore these topics in more detail in future courses.

SEARCHING ALGORITHMS

In this chapter:

- Searching algorithms
- Binary search, linear search
- BFS & DF

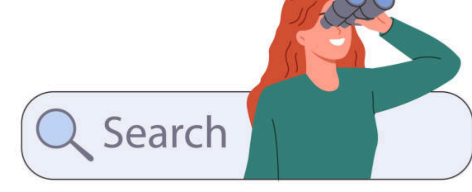
Searching Algorithms

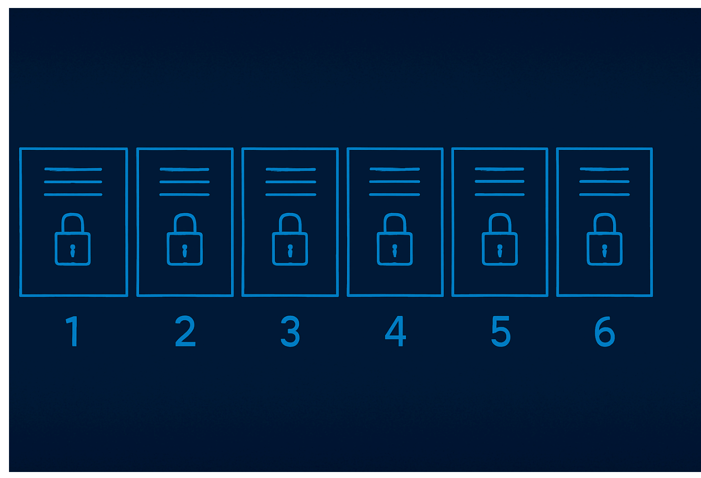
Searching algorithms are essential tools in computer science used to locate specific items within a collection of data. These algorithms are designed to efficiently navigate through data structures to find the desired information, making them fundamental in various applications such as databases, web search engines, and more.

Linear Search

Recall the definition of arrays. We know about the idea of an array, blocks of memory that are consecutive: side-by-side with one another.

You can metaphorically imagine an array like a series of seven blue lockers as follows:

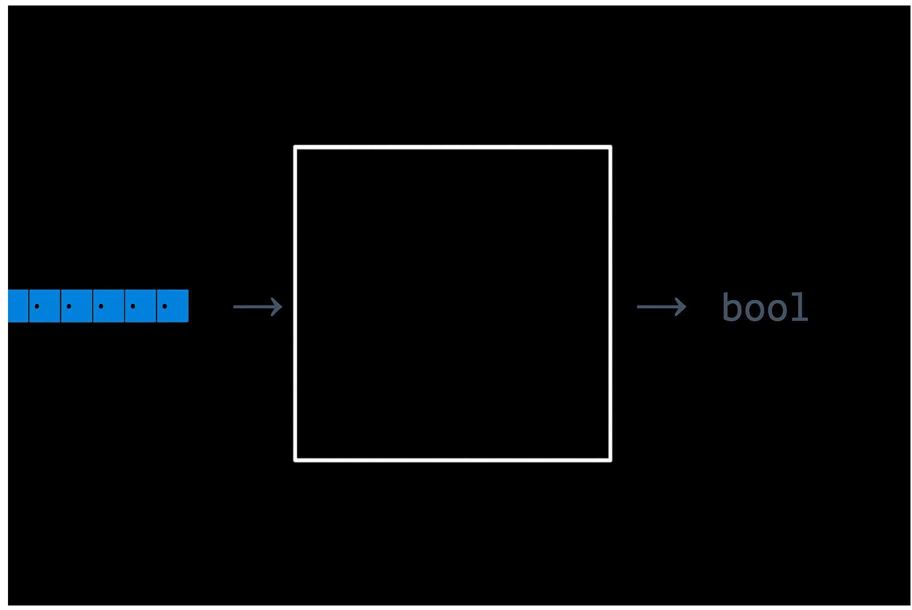




The far-left position is called location 0 or the beginning of the array. The far-right position is location 6 or the end of the array.

We can imagine that we have an essential problem of wanting to know, “Is the number 50 inside an array?” A computer must look at each locker to be able to see if the number 50 is inside. We call this process of finding such a number, character, string, or other item searching.

We can potentially hand our array to an algorithm, wherein our algorithm will search through our lockers to see if the number 50 is behind one of the doors, returning the value true or false.



We can imagine various instructions we might provide our algorithm to undertake this task as follows:

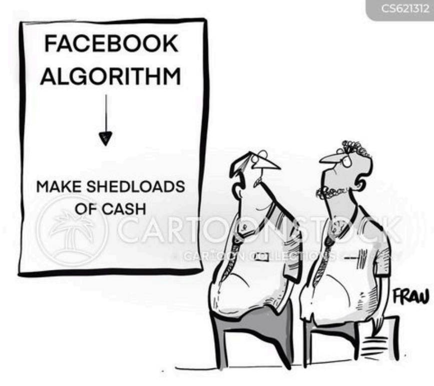
```
for each door from left to right
    if 50 is behind door
        return True
return False
```

Notice that the above instructions are called pseudocode: A human-readable version of the instructions that we could provide the computer.

A computer scientist could translate that pseudocode as follows:

```
for i from 0 to n-1
    if 50 is behind doors[i]
        return true
return false
```

Notice that the above is still not code, but it is a pretty close approximation of what the final code might look like.



Search.c

You can implement linear search by typing code search.c in your terminal window and by writing code as follows:

```
// Implements linear search for integers

#include <stdio.h>

int main(void)
{
    // An array of integers
    int numbers[] = {20, 500, 10, 5, 100, 1, 50};
```

```
    // Take int
    printf("Number: ");
    if (scanf("%d", &n) != 1)
    {
        return 1;
    }
   // Search for number
    for (int i = 0; i < 7; i++)
    {
        if (numbers[i] == n)
        {
            printf("Found\n");
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

Notice that the line beginning with int numbers[] allows us to define the values of each element of the array as we create it. Then, in the for loop, we have an implementation of linear search. return 0 is used to indicate success and exit the program. return 1 is used to exit the program with an error (failure).

We have now implemented linear search ourselves in C!

Phonebook.c

We can combine the ideas of both numbers and strings into a single program. Type code phonebook.c into your terminal window and write code as follows:

```
// Implements a phone book without structs

#include <stdio.h>
#include <string.h>

int main(void)
{
    // Arrays of strings
    char *names[] = {"Yuliia", "David", "John"};
     char *numbers[] = {"+1-617-495-1000", "+1-617-495-1000",
"+1-949-468-2750"};

    // Search for name
    char name[100];
    printf("Name: ");
    if (fgets(name, sizeof(name), stdin) == NULL)
    {
        return 1;
    }

    for (int i = 0; i < 3; i++)
```

```
    {
        if (strcmp(names[i], name) == 0)
        {
            printf("Found %s\n", numbers[i]);
            return 0;
        }
    }
    printf("Not found\n");
    return 1;
}
```

Notice that Yuliia's number begins with +1-617, David's phone number starts with +1-617, and John's number starts with +1-949. Therefore, names[0] is Yuliia, and numbers[0] is Yuliia's number. This code will allow us to search the phonebook for a person's specific number.

Binary Search

Binary search is another search algorithm that could be employed in our task of finding the 50. It divides the problem in half every single time and compares if the solution we are looking for is on the left half of the problem or the right half. It then starts this cycle again and divides the problem into left and right halves until we find the thing we are looking for.

Assuming that the values within the lockers have been arranged from smallest to largest, the pseudocode for binary search would appear as follows:

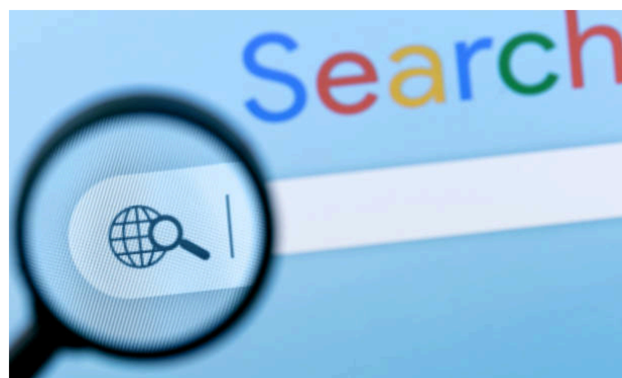
```
if no doors left
    return false
if 50 is behind middle door
    return true
else if 50 < middle door
    search left half
else if 50 > middle door
    search right half
```

Using the nomenclature of code, we can further modify our algorithm as follows:

```
if no doors left
    return false
if 50 is behind doors[middle]
    return true
else if 50 < doors[middle]
    search doors[0] through doors[middle - 1]
else if 50 > doors[middle]
    Search doors[middle + 1] through doors[n - 1]
```

Notice that by looking at this approximation of code, you can nearly imagine what this might look like in actual code.

Case Study: Google Search



Problem: Google needs to find your search term in billions of indexed pages instantly.

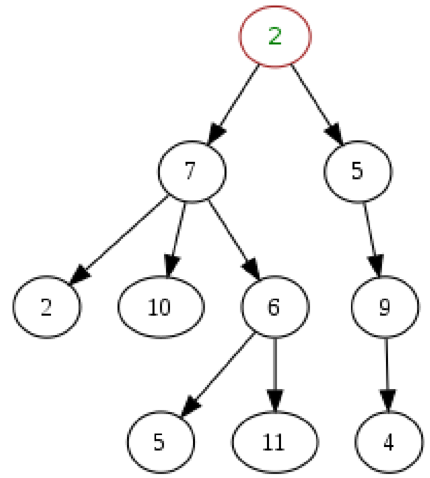
Concept Used: Binary Search (via inverted indexes and ranked retrieval).

Result: Results in <0.2 seconds.

Lesson: Even “simple” algorithms like binary search scale to global systems when layered smartly.

BFS(Breadth-First Search)

So, to better understand these algorithms, we will use the following data tree as an example.



In both BFS and DFS, the first step is to pick a node or vortex to start our search. In this case, I will pick the number 2 at the top. BFS is very straightforward.

We first visit the vortex we chose(2) and we check to see if two have any children, we find out that 2 has 7 and 5 as children.

So we would currently have the following order so far: 2, 7, 5. We then proceed to explore the first child vortex found, which is 7.

We can see that 7 has 3 children: 2, 10, and 6, so now our list would look like this: 2, 7, 5, 2, 10, and 6.

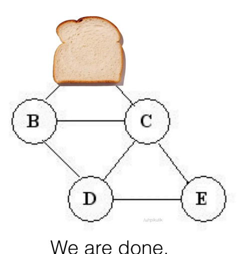
Now we check if the second child(5) has children. We can see that 5 has 9 as a child, so we would add nine to the list: 2, 7, 5, 2, 10, 6, and 9.

We now repeat the same process for the first found children of 7 starting with 2 which has no children so we move to 10 that also has no children so we move to 6 and we find that 6 has 5 and 11 as children so we add them to the list: 7, 5, 2, 10, 6, 9, 5, and 11.

Now we go back and check the child of 5, which is 9. We see that 9 has one child, 4, so we add it to the list: 7, 5, 2, 10, 6, 9, 5, 11, and 4.

Now the next step is to go back to the number 6 and check to see if its children have more children, and we see they have none, so we go to the number 4, and there are also no children, so we are done.

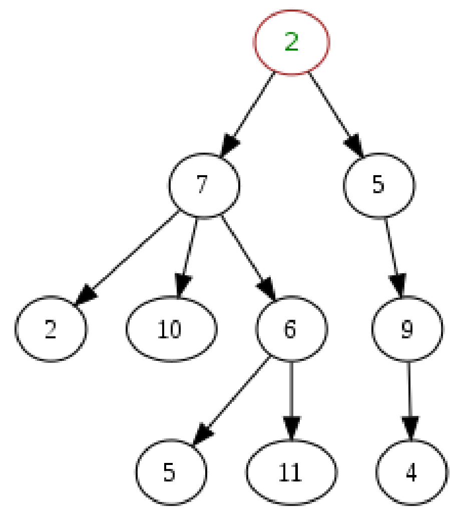
So BFS is pretty easy, right?! It is just like reading any text from left to right, in the simplest case, but in any given tree or dataset, the procedure is the same.



We are done.

DFS(Depth First Search)

To understand this algorithm, I will use the same data tree as an example.



As BFS, we also have to pick an initial vortex to start our search, so I will also pick the number

2 at the top. Now, the difference between BFS and DFS is that DFS goes in a straight line; it doesn’t stop until the deepest part of a vortex is found, hence the name “Depth-first search”. So let's go through each step that DFS takes to complete a search.

I selected the number 2 as my starting point, so the first step is to check if 2 has any children, and we can see that it has 7.

Here is were DFS is different than BFS, instead of checking to see if 2 has more children we stop and we continue with 7, we explore 7 and find out

that 7 has 2 so we stop exploring 7 and we start exploring 2, 2 has nothing so we go back to 7 and we see that 7 has 10, 10 doesn't have anything so we go back to 7 and 7 also has 6.

So far, our list looks like this: 2, 7, 2, 10, and 6.

So next we will explore 6 and find that 6 has 5 we stop exploring 6 and explore 5, we find that 5 doesn't have anything so we go back to 6 and find out that 6 has 11 so we explore 11 we find nothing so now we go back to the first 2 again and our list looks like this now: 2, 7, 2, 10, 6, 5, and 11.

We now explore 2 again and see that it has another child, 5, so we stop exploring 2 and we explore 5 and we find that 5 has 9, so we explore 9 and 9 has 4, we explore 4 and find that 4 doesn't have anything, so we go back until we reach 2 because there are no more children. We explore 2 again and find that 2 has no more children, so we are finished, and our final list looks like this: 2, 7, 2, 10, 6, 5, 11, 5, 9, and 4.

SORTING ALGORITHMS

In this chapter:

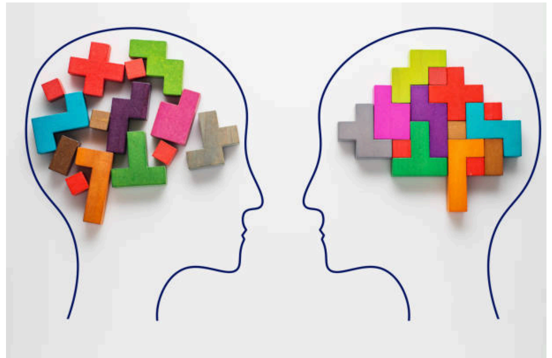
- Sorting algorithms (Iterative & Recursive search)
- Merge sort, quick sort, selection sort and bubble sort

Sorting Algorithms (Iterative/introductory sorts)

Sorting is the act of taking an unsorted list of values and transforming this list into a sorted one.

When a list is sorted, searching that list is far less taxing on the computer. Recall that we can use binary search on a sorted list, but not on an unsorted one.

It turns out that there are many different types of sorting algorithms.



Selection sort

Selection sort is one such sorting algorithm.

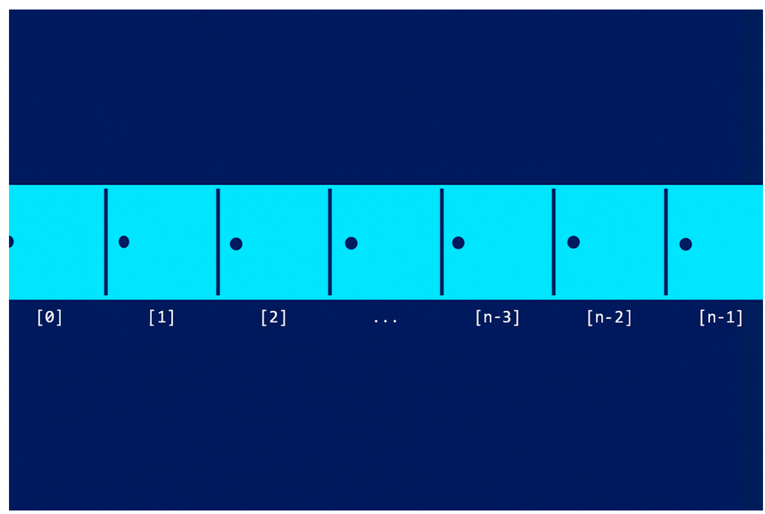
Selection Sort is a comparison-based sorting algorithm. It sorts an array by repeatedly selecting the smallest (or largest) element from the unsorted portion and swapping it with the first unsorted element. This process continues until the entire array is sorted.

First, we find the smallest element and swap it with the first element. This way, we get the smallest element at its correct position.

Then we find the smallest among the remaining elements (or second smallest) and swap it with the second element.

We keep doing this until we get all elements moved to the correct position.

We can represent an array as follows:



The algorithm for selection sort in pseudocode is:

```
for i from 0 to n-1
     Find smallest number between numbers[i] and
numbers[n-1]
    Swap smallest number with numbers[i]
```

Summarizing those steps, the first time iterating through the list took n - 1 steps. The second time, it took $n - 2$ steps. Carrying this logic forward, the steps required could be represented as follows:

$$(n - 1) + (n - 2) + (n - 3) + \dots + 1$$

This could be simplified to $n(n - 1)/2$ or, more simply, $O(n2)$. In the worst-case or upper-bound, selection sort is in the order of $O(n2)$. In the best-case, or lower-bound, selection sort is in the order of $\Omega(n2)$.

Bubble Sort

Bubble sort is another sorting algorithm that works by repeatedly swapping elements to "bubble" larger elements to the end.

The pseudocode for bubble sort is:

```
Repeat n-1 times
    For i from 0 to n-2
        If numbers[i] and numbers[i+1] out of order
            Swap them
    If no swaps
        Quit
```

As we further sort the array, we know more and more of it becomes sorted, so we only need to look at the pairs of numbers that haven’t been sorted yet.

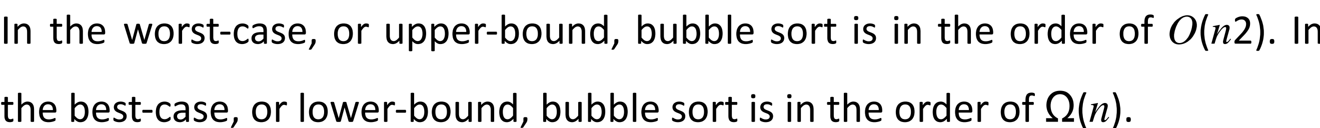
Bubble sort can be analyzed as follows:

$$(n - 1) \times (n - 1)$$

$$n2 - 1n - 1n \; + \; 1$$

$$n2 - 2n \; + \; 1$$

or, more simply $O(n2)$



In the worst-case, or upper-bound, bubble sort is in the order of $O(n2)$. In the best-case, or lower-bound, bubble sort is in the order of $\Omega(n)$.

You can visualize a comparison of these algorithms.

Insertion Sort

Insertion sort is a very simple method to sort numbers in an ascending or descending order. This method follows the incremental method. It can be compared with the technique of how cards are sorted at the time of playing a game.

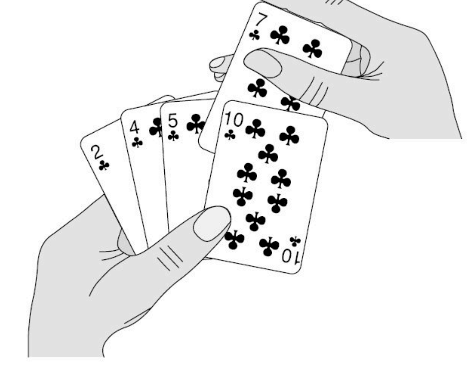
This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained, which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'inserted' in this sorted sub-list has to find its appropriate place, and then it has to be inserted there. Hence the name, insertion sort.

The array is searched sequentially, and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of (n2), where n is the number of items.

Insertion Sort Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

Step 1 − If it is the first element, it is already sorted. return 1;



Step 2 − Pick the next element

Step 3 − Compare with all elements in the sorted sub-list

Step 4 – Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Step 5 – Insert the value

Step 6 – Repeat until the list is sorted

Analysis

The run time of this algorithm is very much dependent on the given input.

If the given numbers are sorted, this algorithm runs in O(n) time. If the given numbers are in reverse order, the algorithm runs in O(n2) time.

Recursion

How could we improve our efficiency in our sorting?

Recursion is a concept within programming where a function calls itself. We saw this earlier when we saw…

```
if no doors left
    Return false
if number behind middle door
```

```
    Return true
else if number < middle door
    Search left half
else if number > middle door
    Search right half
```

Notice that we are calling search on smaller and smaller iterations of this problem.

Similarly, in our pseudocode for Week 0, you can see where recursion was implemented:

```
Pick up phone book
Open to middle of phone book
Look at page
if person is on page
    Call person
else if person is earlier in book
   Open to middle of left half of book
    Go back to line 3
else if person is later in book
    Open to middle of right half of book
     Go back to line 3
else
    Quit
```

This code could have been simplified to highlight its recursive properties as follows:

```
Pick up phone book
Open to middle of phone book
Look at page
if person is on page
    Call person
else if person is earlier in book
    Search left half of book
else if person is later in book
   Search right half of book
else
     Quit
```

Consider how we wanted to create a pyramid structure as follows:

Type code iteration.c into your terminal window and write code as follows:

```
// Draws a pyramid using iteration

#include <stdio.h>

void draw(int n);
```

```
int main(void)
{
    // Get height of pyramid
    int height;
    scanf("%d”, &height");

    // Draw pyramid
    draw(height);
}

void draw(int n)
{
    // Draw pyramid of height n
    for (int i = 0; i < n; i++)
    {
        for (int j = 0; j < i + 1; j++)
        {
            printf("#");
        }
        printf("\n");
    }
}
```

Notice that this code builds the pyramid by looping.

To implement this using recursion, type code iteration.c into your terminal window and write code as follows:

```
// Draws a pyramid using recursion
```

```
#include <stdio.h>

void draw(int n);

int main(void)
{
    // Get height of pyramid
    int height;
    printf("Height: ");
    scanf("%i", &height);

    // Draw pyramid
    draw(height);
}

void draw(int n)
{
    // If nothing to draw
    if (n <= 0)
    {
        return;
    }

    // Draw pyramid of height n - 1
    draw(n - 1);

    // Draw one more row of width n
    for (int i = 0; i < n; i++)
    {
        printf("#");
```

```
    }
    printf("\n");
}
```

Notice that the base case will ensure the code does not run forever. The line if (n <= 0) terminates the recursion because the problem has been solved. Every time, draw calls itself, it calls itself by n-1. At some point, n-1 will equal 0, resulting in the draw function returning, and the program will end.

Sorting Algorithms (Recursive/divide and conquer sorts)

We can now leverage recursion in our quest for a more efficient sort algorithm and implement what is called merge sort, a very efficient sort algorithm.

The pseudocode for merge sort is quite short:

```
If only one number
    Quit
Else
    Sort left half of number
    Sort right half of number
    Merge sorted halves
```

Consider the following list of numbers:

6341

First, merge sort asks, “Is this one number?” The answer is “no,” so the algorithm continues.

6341

Second, merge sort will now split the numbers down the middle (or as close as it can get) and sort the left half of the numbers.

63|41

Third, merge sort would look at these numbers on the left and ask, “Is this one number?” Since the answer is no, it would then split the numbers on the left down the middle.

6|3

Fourth, merge sort will again ask, “Is this one number?” The answer is yes, this time! Therefore, it will quit this task and return to the last task it was running at this point:

63|41

Fifth, merge sort will sort the numbers on the left.

36|41

Now, we return to where we left off in the pseudocode, now that the left side has been sorted. A similar process of steps 3-5 will occur with the right-hand numbers. This will result in:

36|14

Both halves are now sorted. Finally, the algorithm will merge both sides. It will look at the first number on the left and the first number on the right. It will put the smaller number first, then the second smallest. The algorithm will repeat this for all numbers, resulting in:

1346

Merge sort is complete, and the program quits.

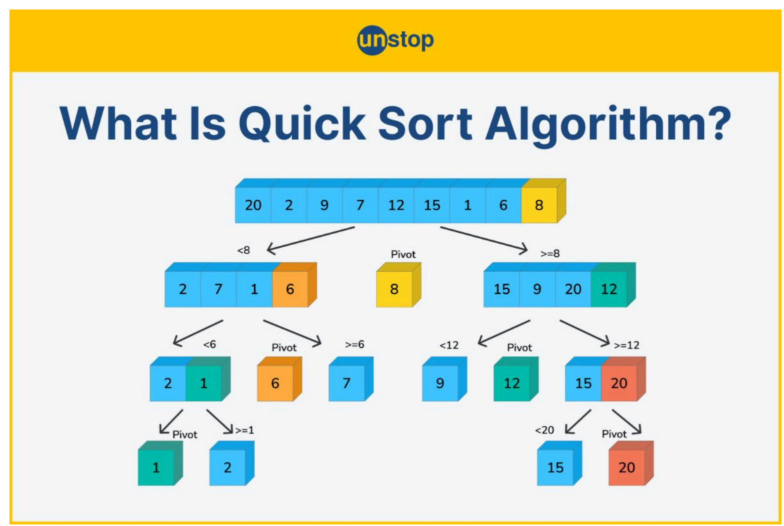
Merge sort is a very efficient sort algorithm with a worst-case of $O(n\log n)$. The best-case is still $\Omega(n\log n)$ because the algorithm still must visit each place in the list. Therefore, merge sort is also $\Theta(n\log n)$ since the best-case and worst-case are the same.

Case Study: Spotify

Spotify sorts your 10 K-song playlist with merge sort every time you hit "shuffle". Stable sort keeps album order intact, and thus you don't get two songs from the same album back-to-back.

Quick sort

QuickSort is one of the best sorting algorithms that follows the divide-and-conquer approach, like Merge Sort, but unlike Merge Sort, this algorithm does in-place sorting. In this article, we will learn how to implement quicksort in C language.



What is a QuickSort Algorithm?

The basic idea behind QuickSort is to select a pivot element from the array and partition the other elements into two sub-arrays according to whether they are less than or greater than the pivot. The sub-arrays are then sorted recursively. This process of partitioning and sorting continues until the entire array is sorted.

QuickSort using C Library

C language provides a library function qsort() that allows you to sort any type of array using a custom comparator method. Now we are going to see how to implement quicksort in C using the library function qsort().

```
// C Program to sort an array using qsort() function in C
#include <stdio.h>
#include <stdlib.h>

// If a should be placed before b, compare function should
// return positive value, if it should be placed after b,
// it should return a negative value. Returns 0 otherwise
int compare(const void* a, const void* b) {
    return (*(int*)a - *(int*)b);
}
```

```c
int main() {
    int arr[] = { 4, 2, 5, 3, 1 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Sorting array using inbuilt quicksort method
    qsort(arr, n, sizeof(int), compare);

    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

    return 0;
}
```

Output

1 2 3 4 5

Complexity Analysis

Time Complexity: $O(n\ logn)$, where n is the size of the array.

Auxiliary Space: $O(log\ n)$, considering auxiliary stack s

MASTERING DATA STRUCTURES

A practical playbook
VOL.2

2025

DATA DRIVEN VENTURES

PRESENTED BY
CAINIAO

CHAPTER

INTRODUCTION TO DATA STRUCTURES

In this chapter:

- You learn what is a data structure
- What is the importance of data structures

Introduction

A data structure is a data storing format. It is the collection of values and the format they are stored in, the relationships between the values in the collection, as well as the operations applied on the data stored in the structure. It is essentially a form of organization in memory.

Abstract data types are those that we can conceptually imagine. When learning about computer science, it’s often useful to begin with this conceptual data.

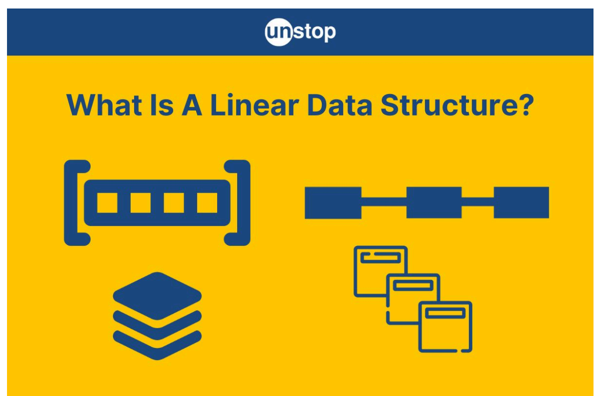
The importance of data structures

- Efficiency: Proper data structures make algorithms run faster (e.g., searching, sorting, and inserting).
- Memory Management: They help us use memory wisely and avoid waste.
- Scalability: With the right structure, programs handle large datasets without slowing down.
- Problem Solving: Certain problems are much easier to solve with specific data structures.

Learning Suggestions

1. **Use analogies/do it physically:** Using sticky notes or cards to represent linked cards and stacks can help visual learners understand more about how it works. It is also possible to use analogies that work in the same way, for example, the LIFO can be represented by a pile of dishes in the sink, the last plate to go in is the first that is out. Whereas LIFO can represent a queue to watch a movie in the theatre, the first person in the queue is the first to pay and, inherently, the first out of the queue.
2. **Code it:** Although programming languages (notably high-level ones) already handle data structures, you may ask yourself in this case, “Then what’s the point of coding it myself?” It is still great to not only practice coding, but also to understand how the computer runs the algorithm. For beginners, Python is recommended.

LINEAR DATA STRUCTURES



In this chapter:

- You learn what is a linear data structure
- The types of linear data structures
- How they work and their use cases

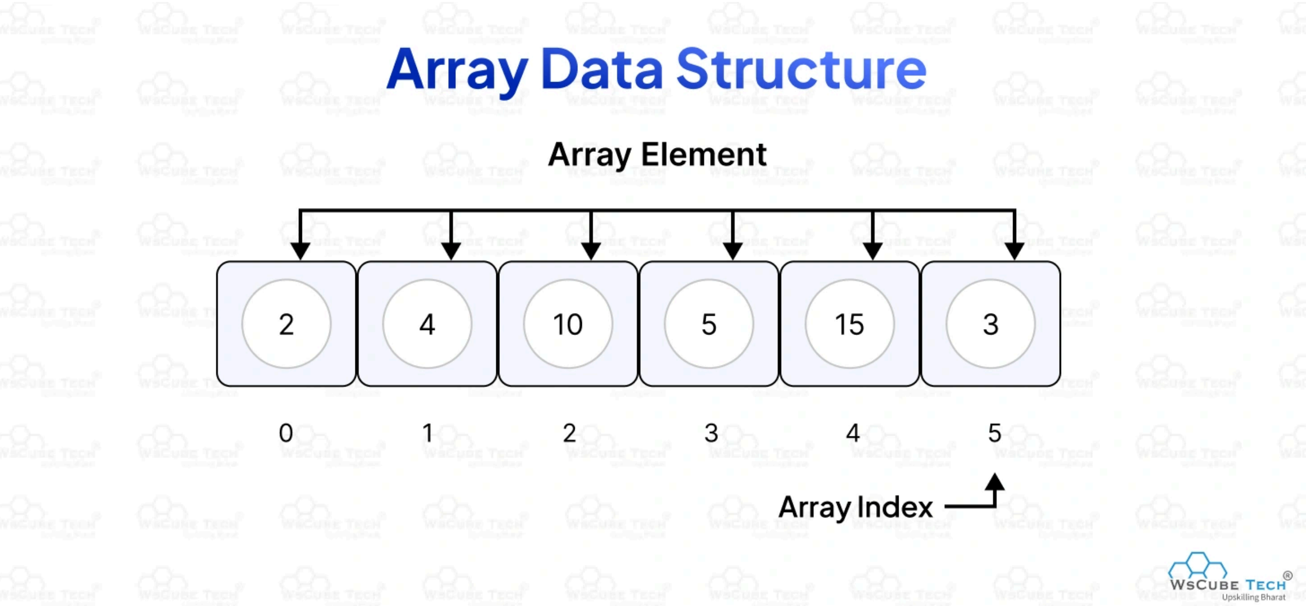
Linear Data Structures

A data structure in which data elements are arranged sequentially or linearly, where each element is attached to its previous and next adjacent elements, is called a linear data structure.

Static Data Structure

A static data structure has a fixed memory size. It is easier to access the elements in a static data structure (e.g. Array).

Array



An array is a data structure that stores a collection of values where each value is referenced using an index or a key. It is a fundamental and linear

data structure, using which we build other data structures like Stack, Queue, Deque, Graph, Hash Table, etc.

Example: array[a, b, c] and their keys are 1, 2, 3, respectively.

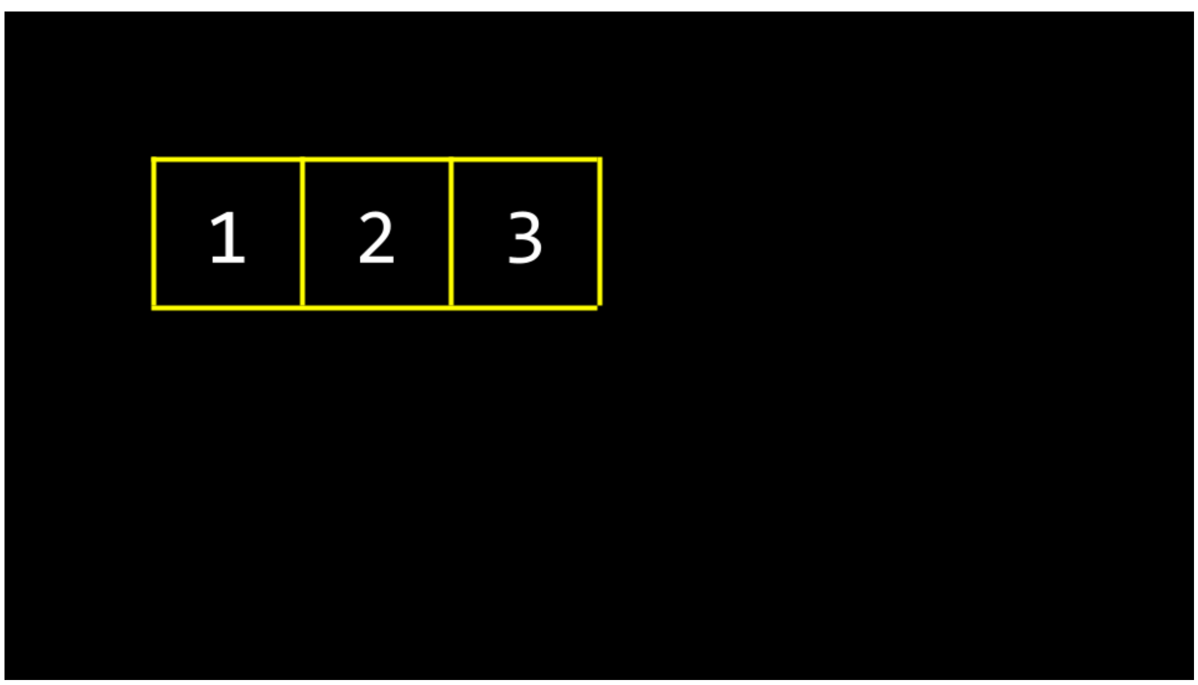
It offers mainly the following advantages over other data structures.

1. Random Access: i-th item can be accessed in O(1) Time as we have the base address and every item or reference is of the same size.
2. Cache Friendliness: Since items/references are stored at contiguous locations, we get the advantage of locality of reference.

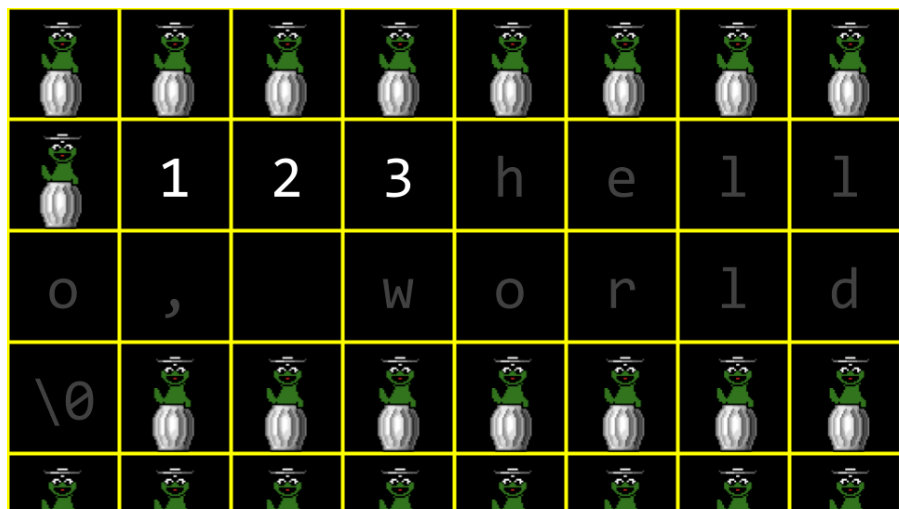
Visualization of an array:

An array is a block of contiguous memory.

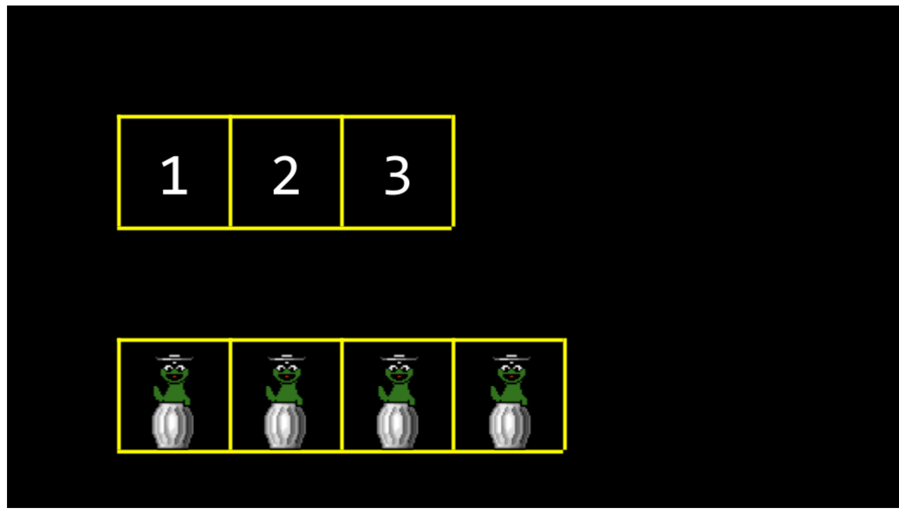
You might imagine an array as follows:



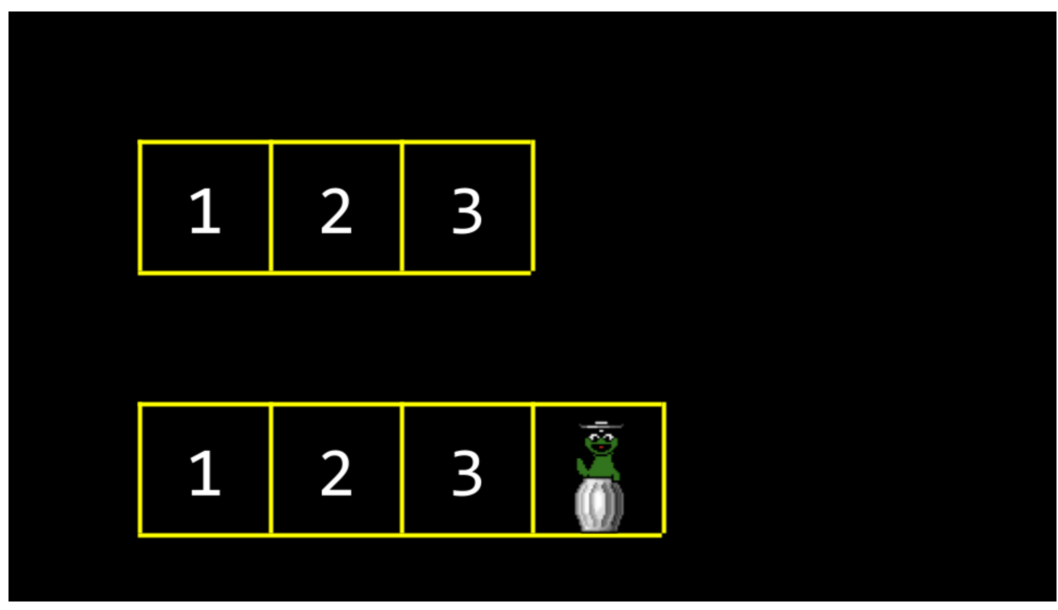
In memory, there are other values being stored by other programs, functions, and variables. Many of these may be unused garbage values that were utilized at one point but are available now for use.



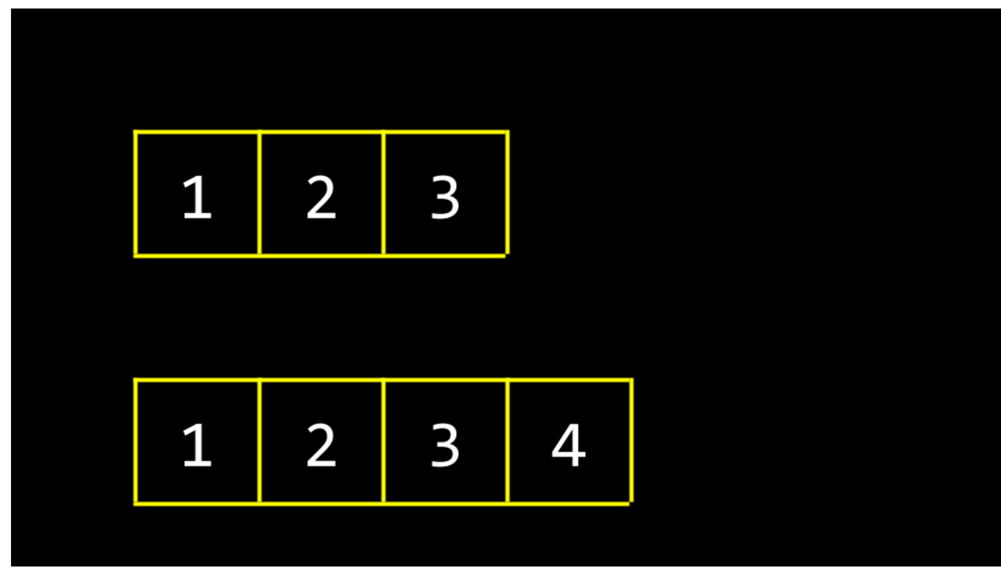
Imagine you wanted to store a fourth value, 4, in our array. What would be needed is to allocate a new area of memory and move the old array to a new one. Initially, this new area of memory would be populated with garbage values.



As values are added to this new area of memory, old garbage values would be overwritten.

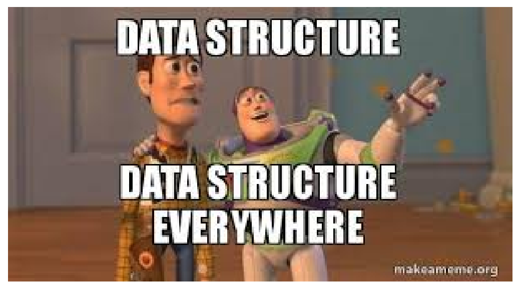


Eventually, all old garbage values would be overwritten with our new data.



One of the drawbacks of this approach is that it's a bad design: Every time we add a number, we have to copy the array item by item.

Wouldn't it be nice if we were able to put the 4 somewhere else in memory? By definition, this would no longer be an array because 4 would no longer be in contiguous memory. How could we connect different locations in memory?



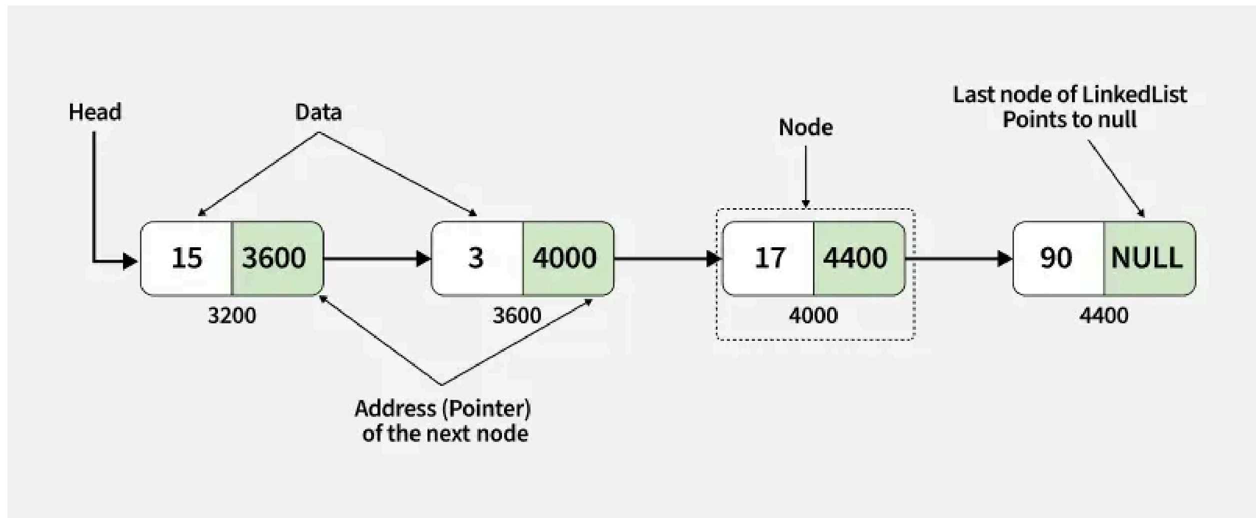
Dynamic Data Structure

That’s where dynamic data structures come in. In a dynamic data structure, the size is not fixed. It can be randomly updated during the runtime, which may be considered efficient concerning the memory (space) complexity of the code (e.g. Queue, Stack, linked lists).

Linked list

A linked list is one of the most powerful data structures within C. A linked list allows you to include values that are located in varying areas of memory. Further, they allow you to dynamically grow and shrink the list as you desire.

Linked lists are linear data structures made up of a series of connected elements called nodes. Unlike arrays, linked lists do not store data in contiguous memory. Instead, each node stores data and a pointer (link) to the next node in the sequence, with the last node having null as its pointer.



You can think of a linked list like a treasure hunt: every clue (node) contains the location of the next clue, but you don't know the whole route in advance.

Structure of a node:

```
typedef struct node
{
    int data;           // Value stored in the node
    struct node *next;  // Pointer to the next node
} node;
```

In this example, data holds the node's value, and next points to the next node in memory.

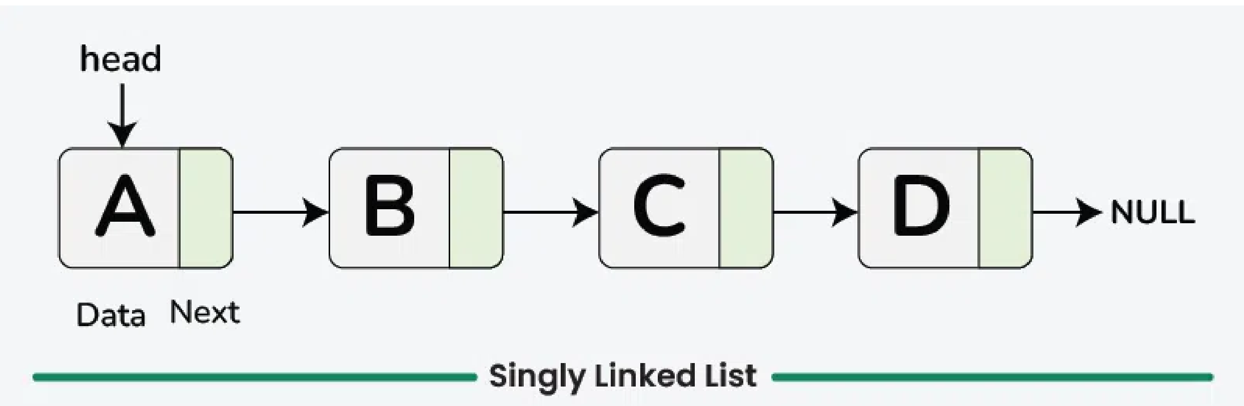
If next is NULL, it means the list has ended.

Main operations in a linked list:

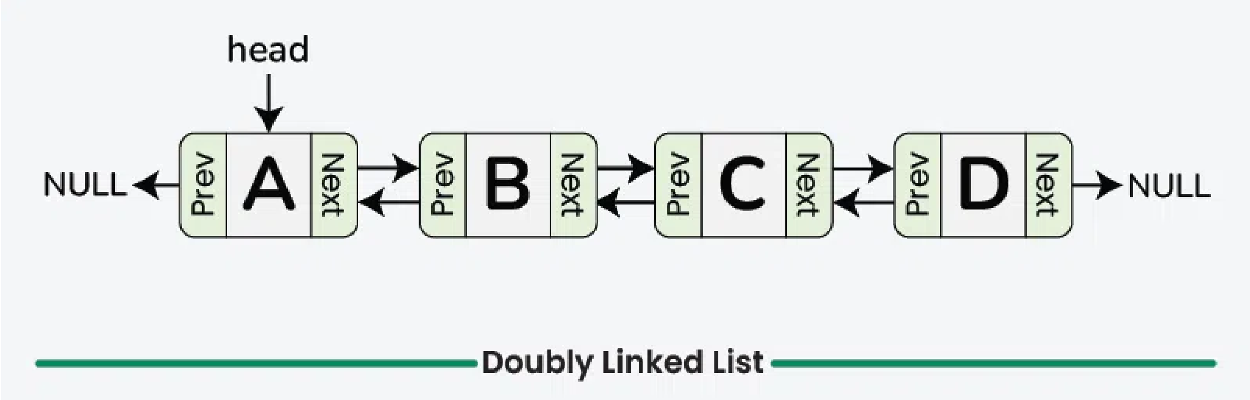
- Insertion: Add a node at the beginning, end, or middle.
- Deletion: Remove a node from the list.
- Traversal: Visit each node starting from the head (first node).

Types of linked lists:

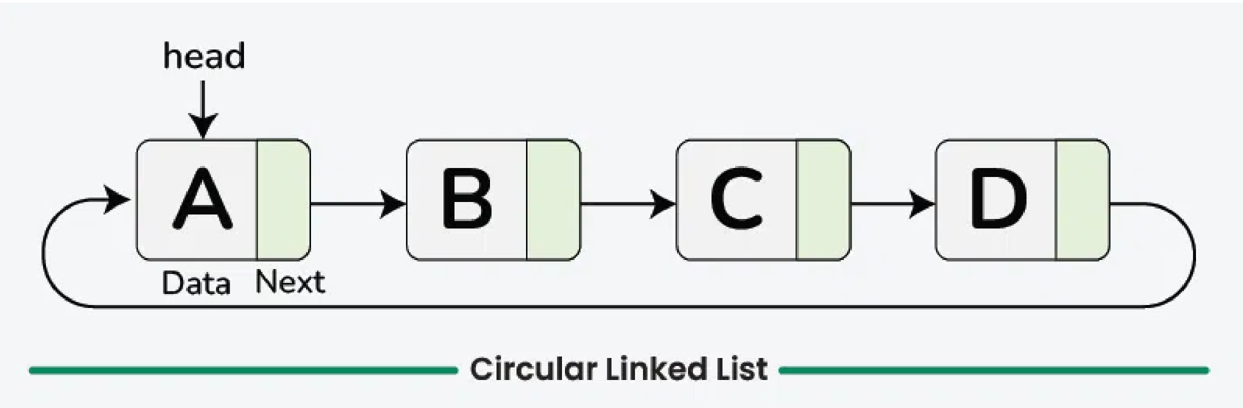
1. Singly Linked List: Each node points only to the next node.



2. Doubly Linked List: Each node points to both the next and previous nodes.



3. Circular Linked List: The last node links back to the first node.



Advantages of using a linked list:

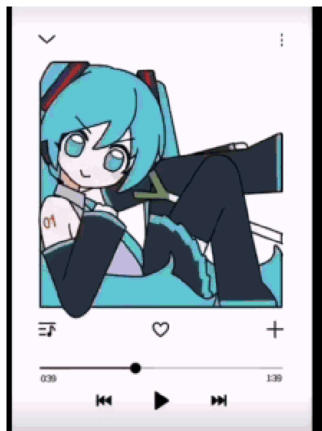
- Flexible size (can grow or shrink easily).
- Efficient insertions and deletions (no shifting like arrays).

Disadvantages of using a linked list:

- No direct access (must traverse from the start).
- Uses extra memory for pointers.

Real-world examples:

Music playlist shuffle: each song node points to the next.



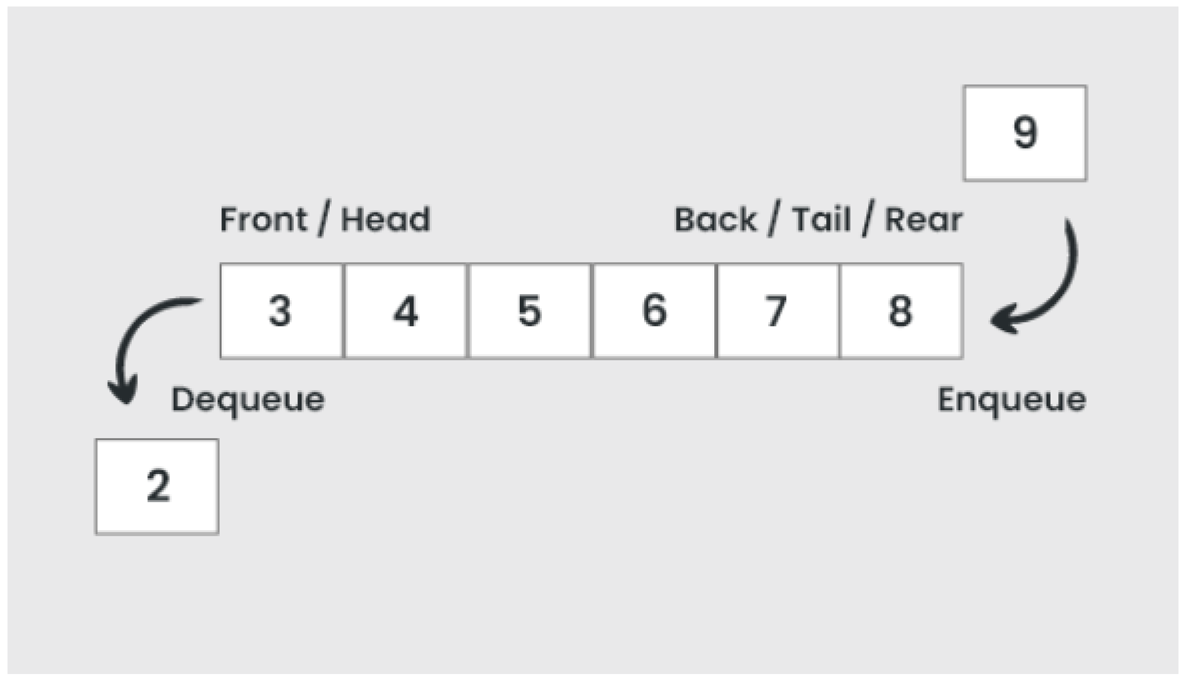
Swiping on a dating app: Each profile swap points to the next profile.

Queues(FIFO)

After understanding linked lists, it becomes clear how flexible and efficient they are for adding and removing data without worrying about contiguous memory. This same idea of connecting elements in sequence can be used to

model real-world situations where order matters, such as people waiting in line.

This brings us to queues, abstract data structures that have specific properties. Namely, they are FIFO or “first in, first out.” You can imagine yourself in a line for a ride at an amusement park. The first person in the line gets to go on the ride first. The last person gets to go on the ride last.



Queues have two main operations:

- Enqueue: Add an element to the back of the queue.
- Dequeue: Remove an element from the front of the queue.

They're used whenever order matters: like printing tasks, CPU scheduling, or serving customers.

In code, you can imagine a queue as follows:

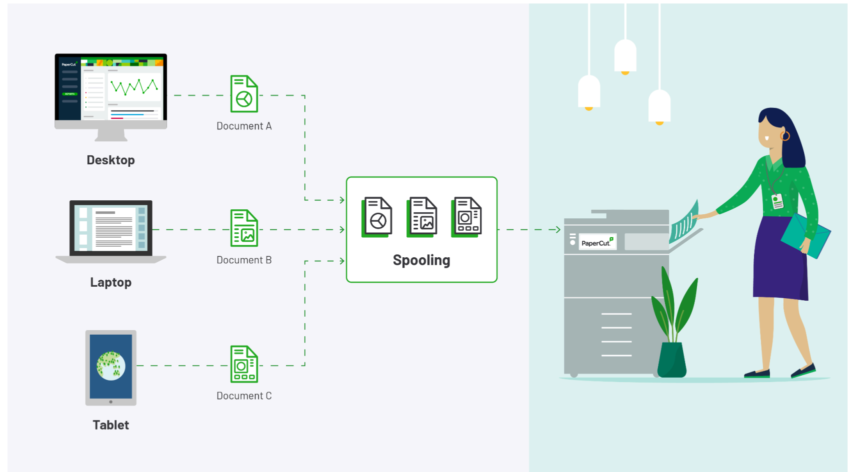
```
const int CAPACITY = 50;   // Maximum number of people the queue can hold

typedef struct
{
      person people[CAPACITY];   // Array to store elements (people)
     int size;                  // Number of people currently in the queue
}
queue;
```

This structure defines a queue with a fixed capacity of 50. The people array holds the data, and size keeps track of how many elements are currently stored.

Real-world examples:

- Printer Spool: where if a printer receives many documents at once, it will print them in the order it was received, First in First out.



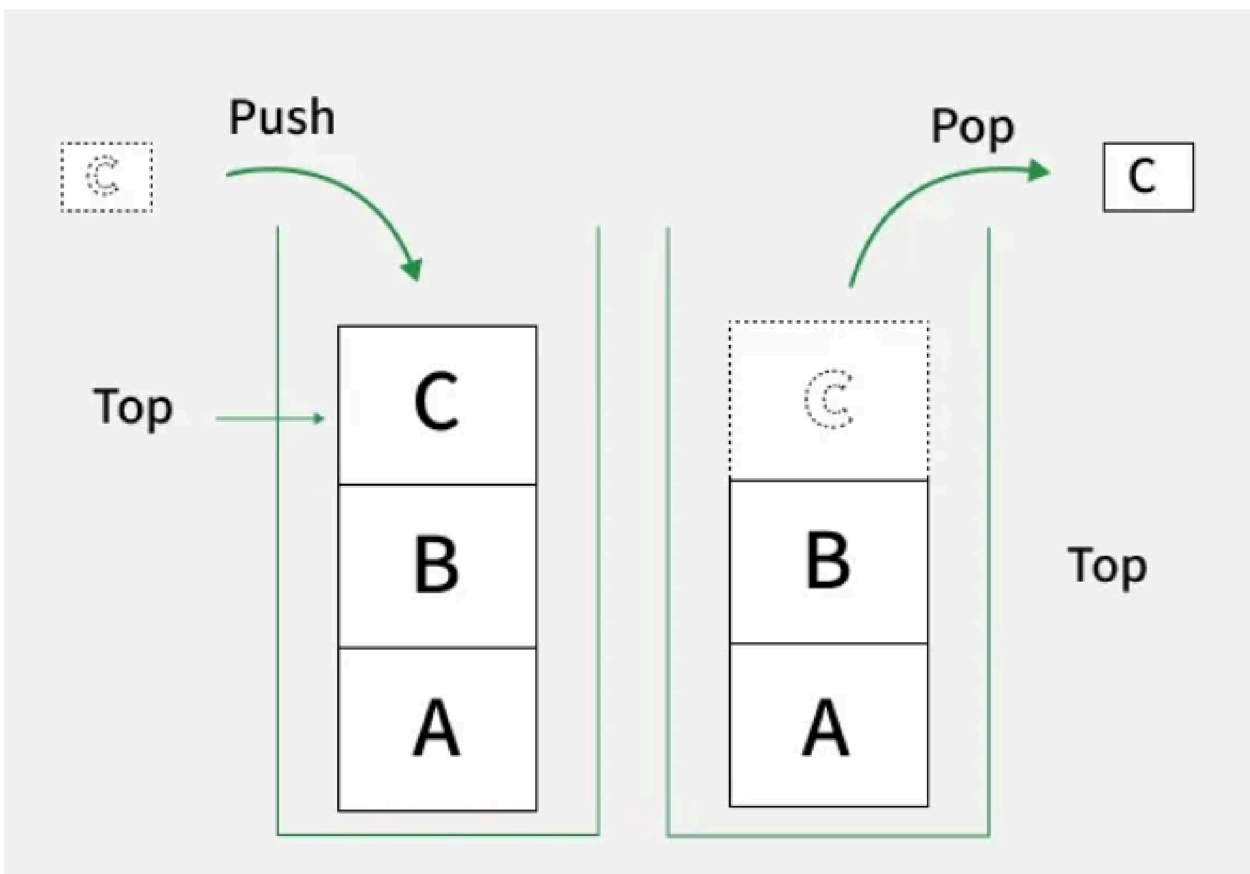
- Chat Messages: FIFO helps keep the conversation in the exact order people typed. The earliest messages are handled first.

Stacks(LIFO)

While queues remove items in the same order they were added, some situations require the opposite. For example, when you press "undo" in a text editor, the last change you made is the first one reversed.

Queues contrast with a stack. Fundamentally, the properties of a stack are different from those of a queue. Specifically, it is LIFO or "last in, first out."

Just like stacking trays in a dining hall, the tray that is placed on top is the first one to be taken off.



Stacks have two main operations:

- Push: Add an element to the top of the stack.
- Pop: Remove an element from the top of the stack.

In code, you might imagine a stack as follows:

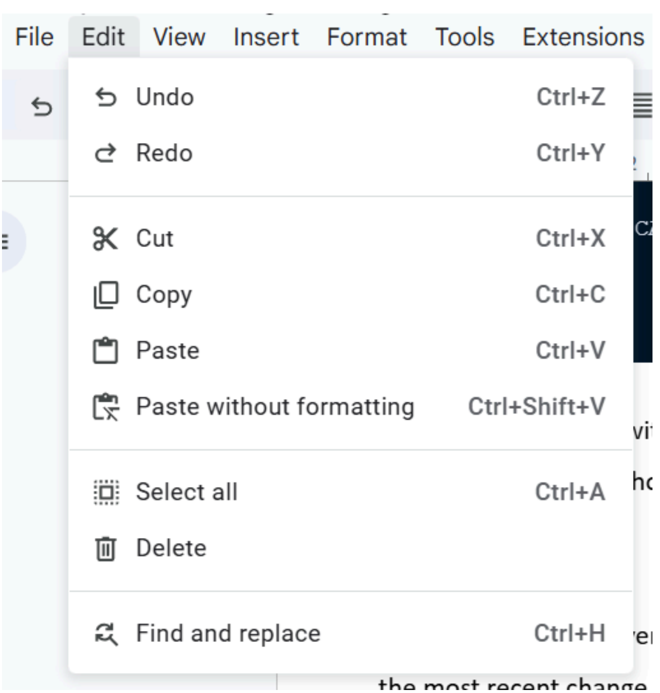
```
const int CAPACITY = 50;
typedef struct
```

```
{
    person people[CAPACITY];
    int size;
}
stack;
```

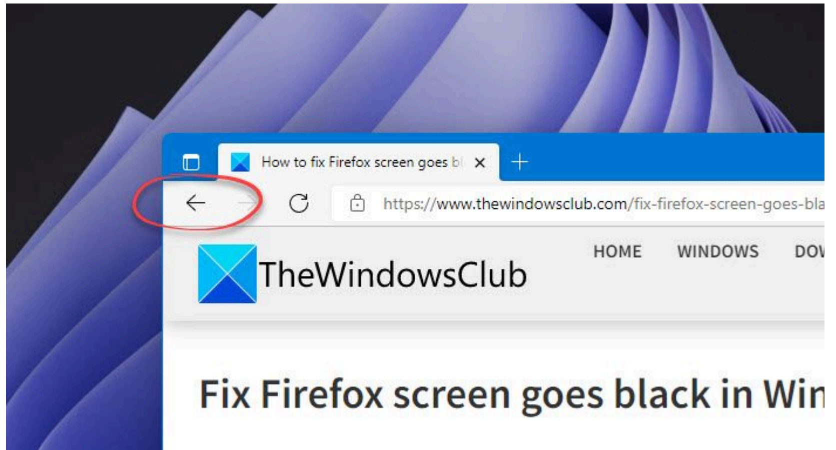
This defines a queue with a fixed capacity of 50. The people array holds the data, and size records how many elements are currently in the queue.

Real-world examples:

- Undo Button: Every edit is pushed onto a stack; pressing undo pops the most recent change.

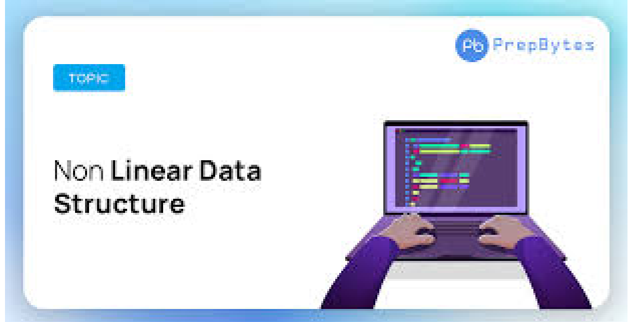


- Browser Back Button: each new page push; back button pops to the previous page.



CHAPTER

NON-LINEAR DATA STRUCTURES



In this chapter:

- You learn what is a non-linear data structure
- The types of non-linear data structures
- How they work and their use cases

Non-linear Data Structures

Linear data structures such as arrays, linked lists, stacks, and queues organize data in a straight line, where each element is connected to the next. They're efficient for managing ordered data, like processing tasks one by one or storing items sequentially.

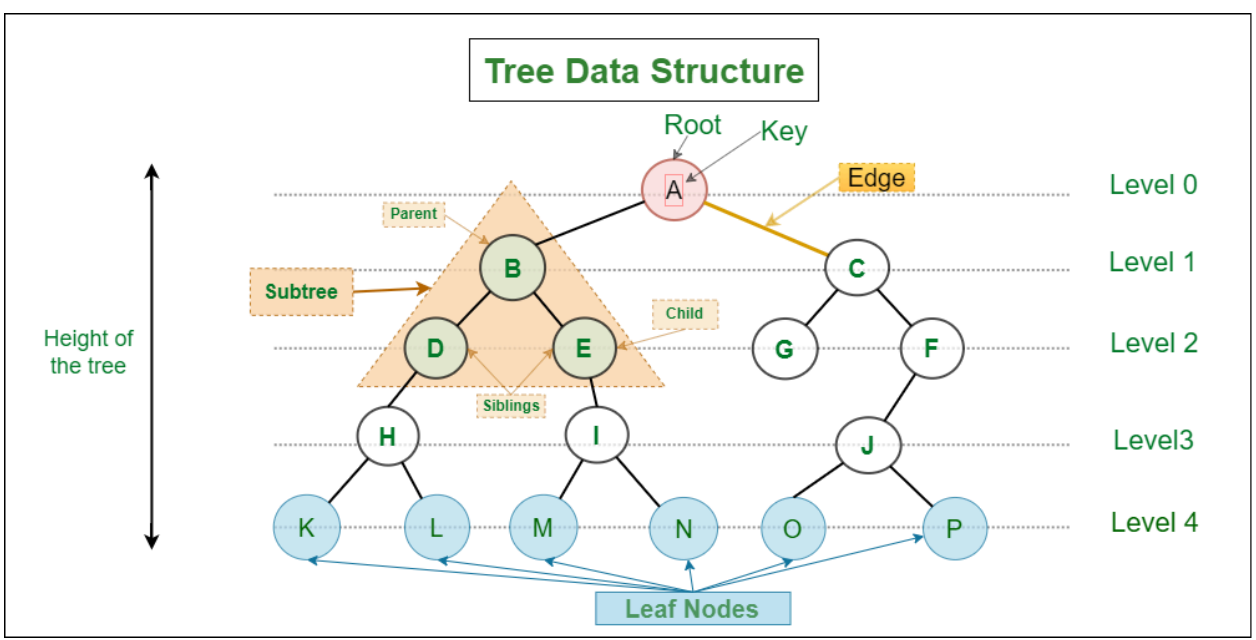
However, many real-world problems don't follow a simple, one-dimensional order. For example, an organization chart, a file system, or a map of connected cities all involve relationships that branch or interconnect in multiple directions.

To represent these complex connections, we use non-linear data structures like trees and graphs. These structures allow data to grow in multiple paths or form networks, making them essential for modelling hierarchies, relationships, and networks in computing. In non-linear data structures, each element can connect to multiple other elements. That means you can't just traverse them in one single direction like a line. The two main types are trees and graphs.

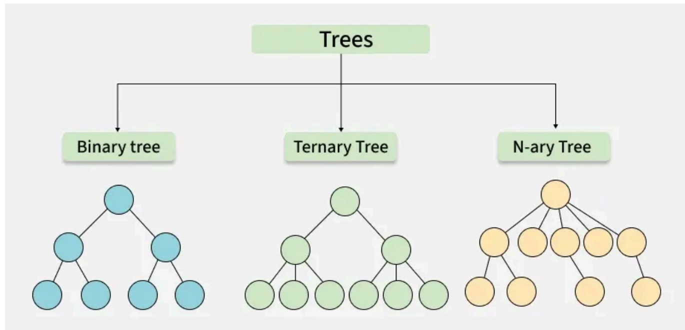
Trees

Trees are non-linear data structures used to represent hierarchical relationships. Each element in a tree is called a node, and the connections between them are called edges. A tree starts with a single root node, which branches out into child nodes. Nodes with no children are called leaves.

You can think of it like a family tree: one ancestor (root) with children and grandchildren branching below.



Types of trees:



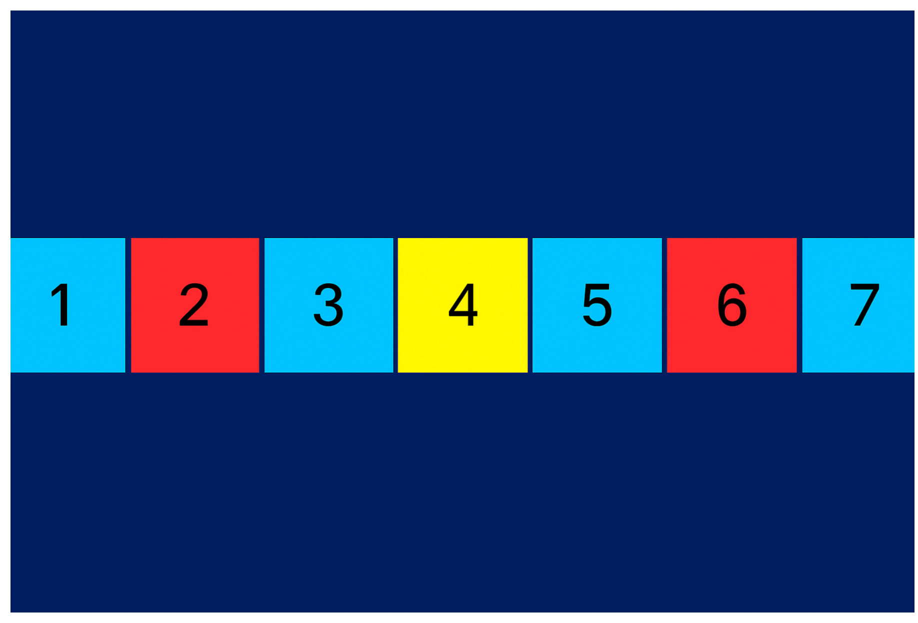
- Binary tree: a tree data structure where each node has at most two children. These two children are usually referred to as the left child and the right child.
- Ternary tree: a tree data structure in which each node has at most three children. These two children are usually referred to as the left child, mid child and right child.
- N-ary tree: it is a generalization of a binary tree, such that each node can have at most N children.

Binary search trees

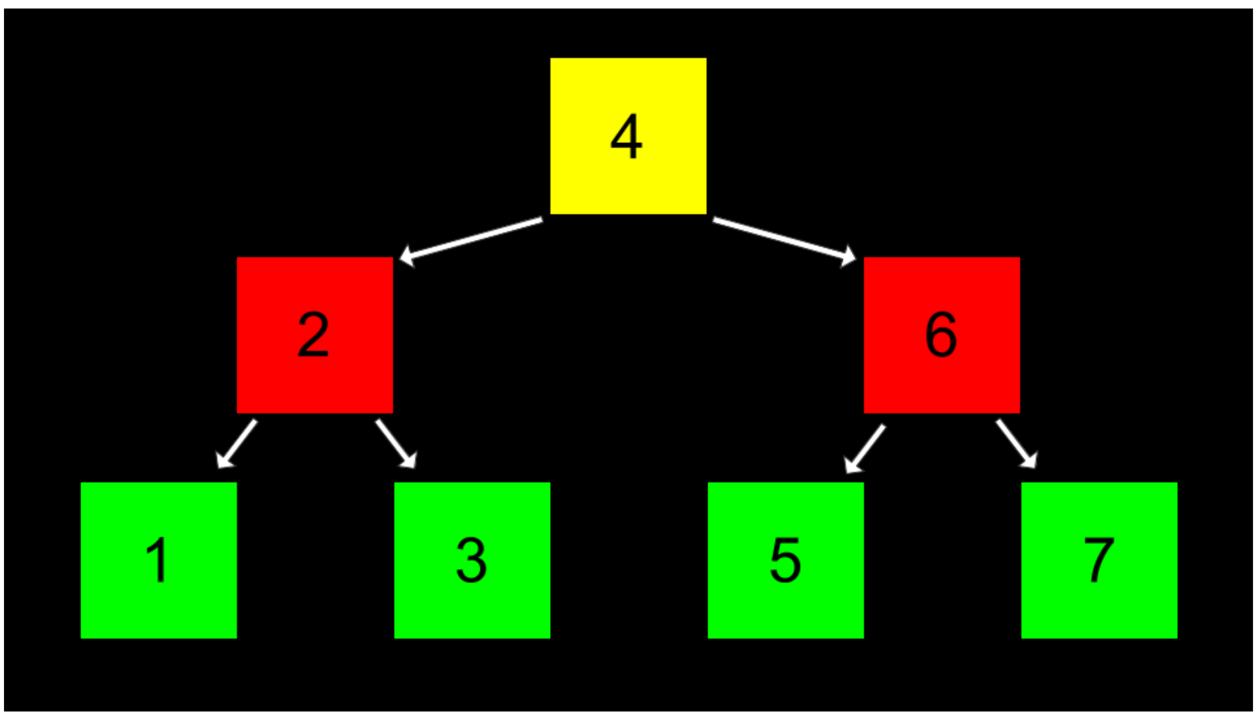
Arrays offer contiguous memory that can be searched quickly. Arrays also offered the opportunity to engage in binary search. Could we combine the best of both arrays and linked lists?

Binary search trees are another data structure that can be used to store data more efficiently so that it can be searched and retrieved. You can imagine a sorted sequence of numbers.

Imagine then that the center value becomes the top of a tree. Those that are less than this value are placed to the left. Those values that are more than this value are to the right.



Pointers can then be used to point to the correct location of each area of memory, such that each of these nodes can be connected.



In code, this can be implemented as follows.

```
// Implements a list of numbers as a binary search tree

#include <stdio.h>
#include <stdlib.h>

// Represents a node
typedef struct node
{
    int number;
    struct node *left;
    struct node *right;
}
node;

void free_tree(node *root);
void print_tree(node *root);

int main(void)
{
    // Tree of size 0
    node *tree = NULL;

    // Add number to list
    node *n = malloc(sizeof(node));
    if (n == NULL)
    {
        return 1;
    }
```

```
    n->number = 2;
    n->left = NULL;
    n->right = NULL;
    tree = n;

    // Add number to list
    n = malloc(sizeof(node));
    if (n == NULL)
    {
        free_tree(tree);
        return 1;
    }
    n->number = 1;
    n->left = NULL;
    n->right = NULL;
    tree->left = n;

    // Add number to list
    n = malloc(sizeof(node));
    if (n == NULL)
    {
        free_tree(tree);
        return 1;
    }
    n->number = 3;
    n->left = NULL;
    n->right = NULL;
    tree->right = n;

    // Print tree
    print_tree(tree);
```

```
    // Free tree
    free_tree(tree);
    return 0;
}

void free_tree(node *root)
{
    if (root == NULL)
    {
        return;
    }
    free_tree(root->left);
    free_tree(root->right);
    free(root);
}

void print_tree(node *root)
{
    if (root == NULL)
    {
        return;
    }
    print_tree(root->left);
    printf("%i\n", root->number);
  print_tree(root->right);
}
```

Notice this search function begins by going to the location of the tree. Then, it uses recursion to search for numbers. The free_tree function recursively frees the tree. print_tree recursively prints the tree.

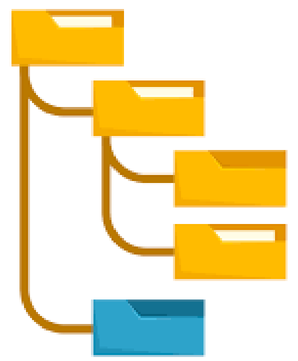
So, in summary, a binary search tree has 4 very distinguishable characteristics:

1. Hierarchical Structure: A BST is composed of nodes, each having up to two children, forming a tree-like hierarchy with a single root node at the top.
2. Ordering Property: For every node in the BST, all values in the left subtree are smaller, and all values in the right subtree are larger than the node's value.
3. Efficient Operations: In a balanced BST, operations like search, insertion, and deletion can be performed in O(log n) time.
4. Recursive Nature: Each left or right subtree of a node in a BST is itself a BST, allowing recursive algorithms to naturally process the tree.

Real-world examples:

- A file directory on your computer:

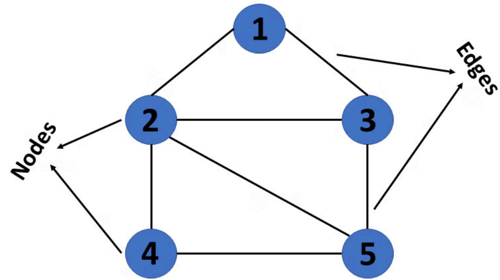
  Where the "Documents" folder might be the root. Inside it are child folders like "School" or "Projects". Inside "School", there might be more files.



Graphs

Trees are powerful for representing hierarchical relationships, like family trees or folder structures. However, not all relationships fit neatly into a hierarchy; sometimes connections form networks instead.

For example, cities connected by roads, or people connected on social media, can have multiple links between them. To represent these more complex and flexible connections, we use a graph. A graph generalizes the idea of a tree; instead of each node having just one parent, any node can connect to any other node.



A graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes, and the edges are lines or arcs that connect any two nodes in the graph. It is represented as G(V, E), where:

- V is the set of vertices (nodes)
- E is the set of edges (connections between nodes)

Imagine a game of football as a web of connections, where players are the nodes and their interactions on the field are the edges. This web of connections is exactly what a graph data structure represents: a system of objects connected to each other.

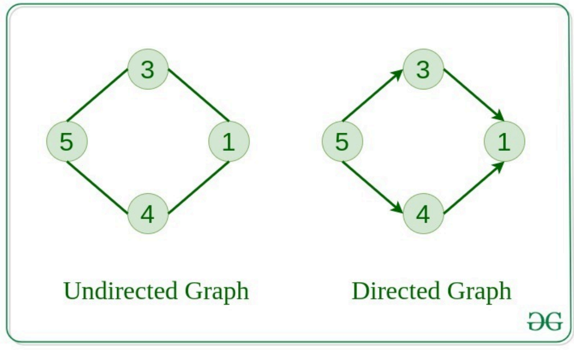
Components of Graph Data Structure:

- Vertices: Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabeled.
- Edges: Edges are drawn or used to connect two nodes of the graph. It can be ordered by a pair of nodes in a directed graph. Edges can

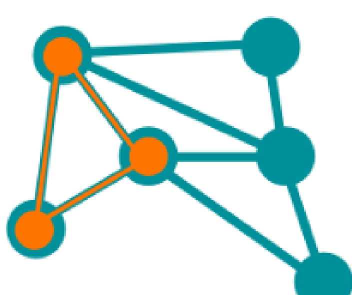
connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabelled.

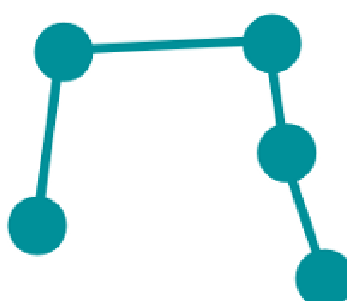
Types Of Graphs:

1. Undirected Graph: A graph in which edges do not have any direction. That is, the nodes are unordered pairs in the definition of every edge.

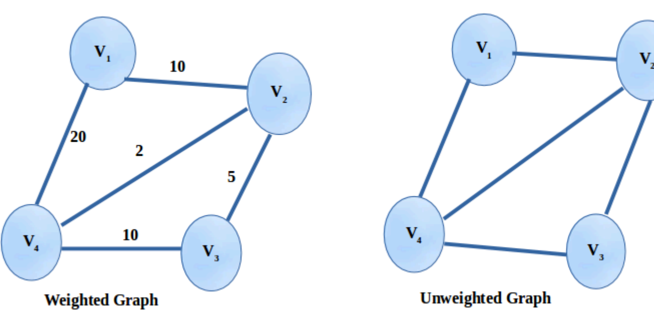


2. Directed Graph: A graph in which the edge has direction. That is, the nodes are ordered pairs in the definition of every edge.
3. Cyclic Graph: A graph containing at least one cycle is known as a Cyclic graph.
4. Directed Acyclic Graph: A Directed Graph that does not contain any cycle.





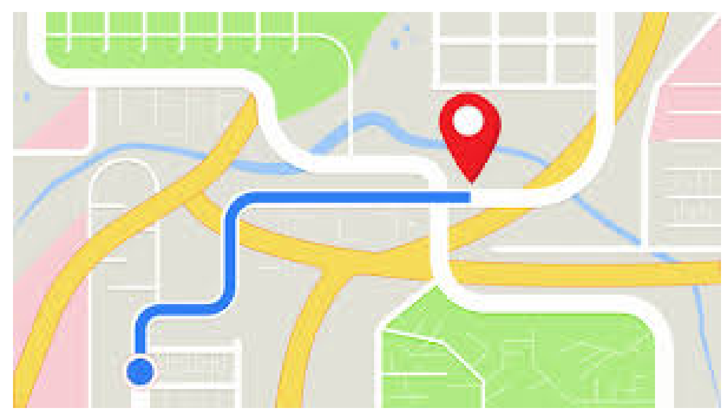
5. Weighted Graph: A graph in which the edges are already specified with a suitable weight is known as a weighted graph.
6. Unweighted graphs: A graph where all edges are treated equally, with no extra values like distance or cost.



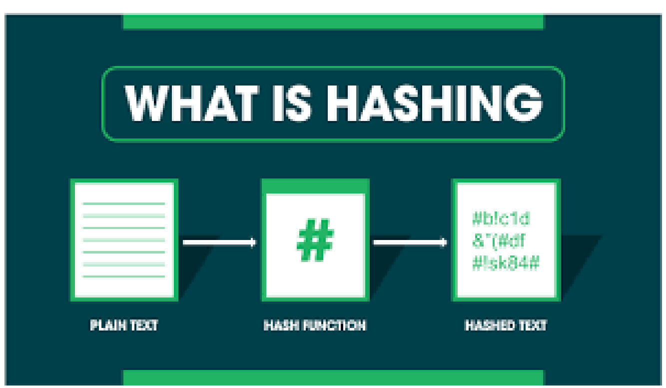
Real-world examples:

- a social network: where each person is a vertex, and if two people are friends, there's an edge connecting them.

- Another example is Google Maps: which uses graphs to represent cities and roads; intersections are vertices, and roads are edges.



HASH-BASED DATA STRUCTURES



In this chapter:

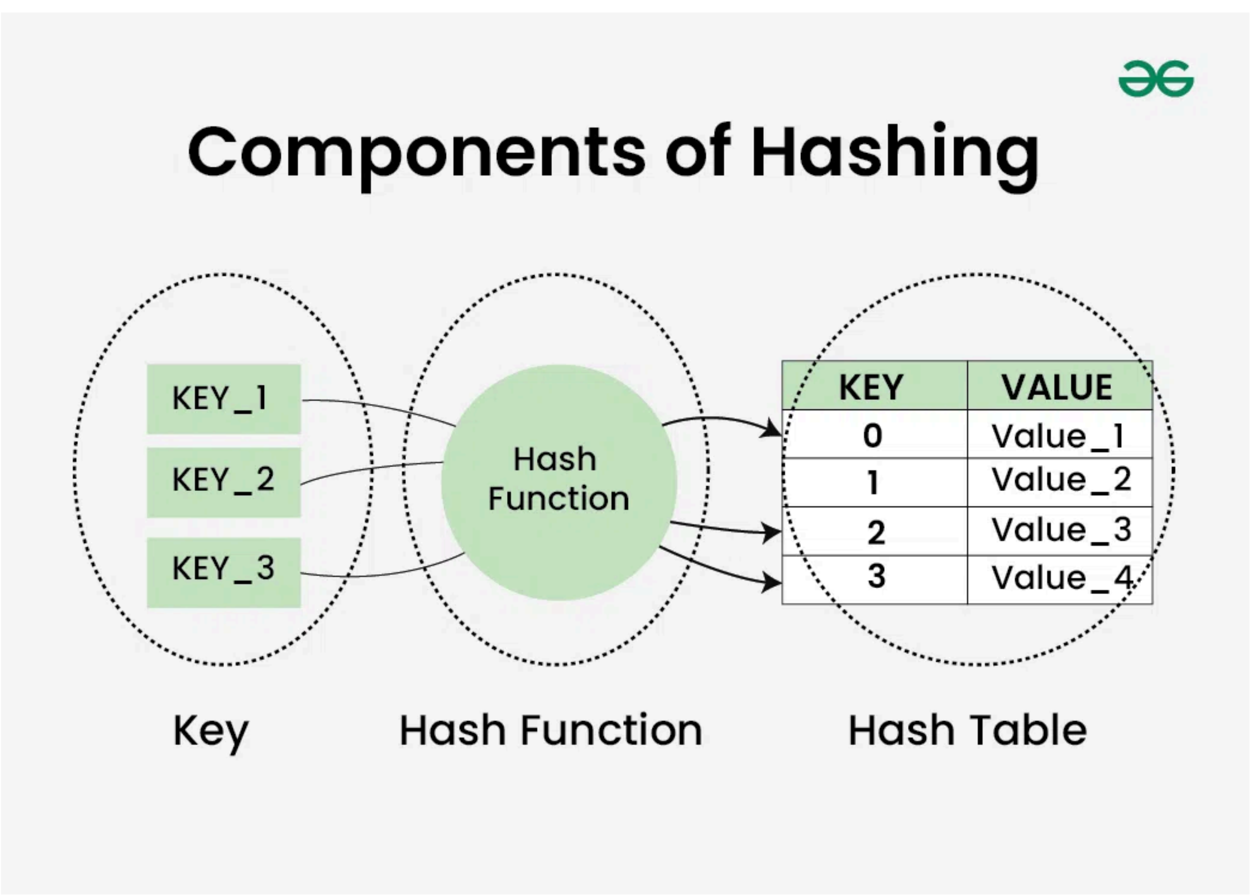
- You learn what is hashing
- What are hash functions and hash tables, and how do they work

Hash-Based Data Structures

Non-linear data structures such as trees and graphs allow us to represent complex relationships between data, including hierarchies, networks, and connections. They are very useful for organizing data for traversal, searching, and analyzing relationships. Sometimes, however, we only need to store and retrieve individual items as quickly as possible.

This is where hashing becomes important. Hashing provides a way to jump directly to the location of a value using a simple calculation called a hash function. This makes operations like search, insertion, and deletion extremely fast, often in constant time (O(1)).

Hashing is the idea of taking a value and being able to output a value that becomes a shortcut to it later. Imagine a library where each book has a number based on the first letter of its title. Instead of scanning all books, you can go directly to the section corresponding to that number.



A hash function is an algorithm that reduces a larger value to something small and predictable. Generally, this function takes in an item you wish to add to your hash table and returns an integer representing the array index in which the item should be placed.

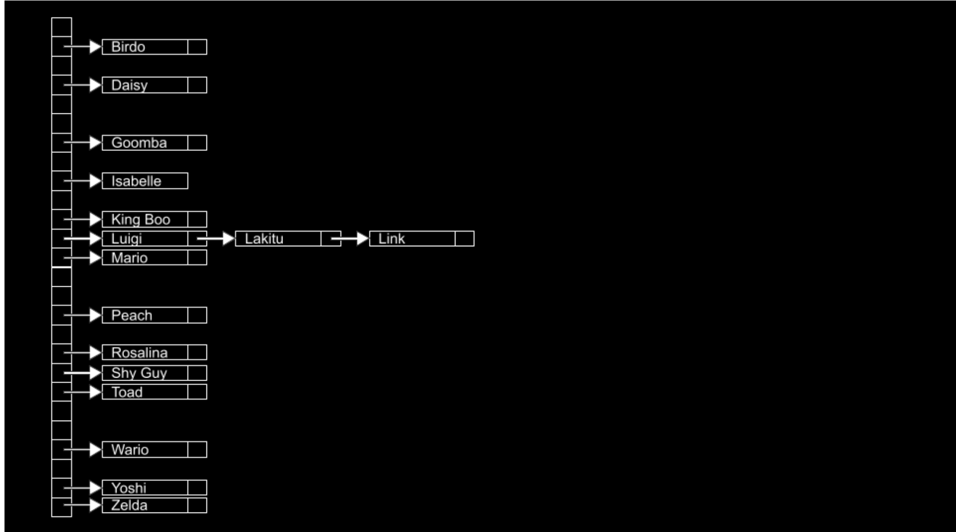
Hash Table

A hash table is a fantastic combination of both arrays and linked lists. When implemented in code, a hash table is an array of pointers to nodes. It uses hashing to store and retrieve data efficiently. Think of it as an array where

the index is determined by a hash function. The hash function computes the index where an item should be stored.

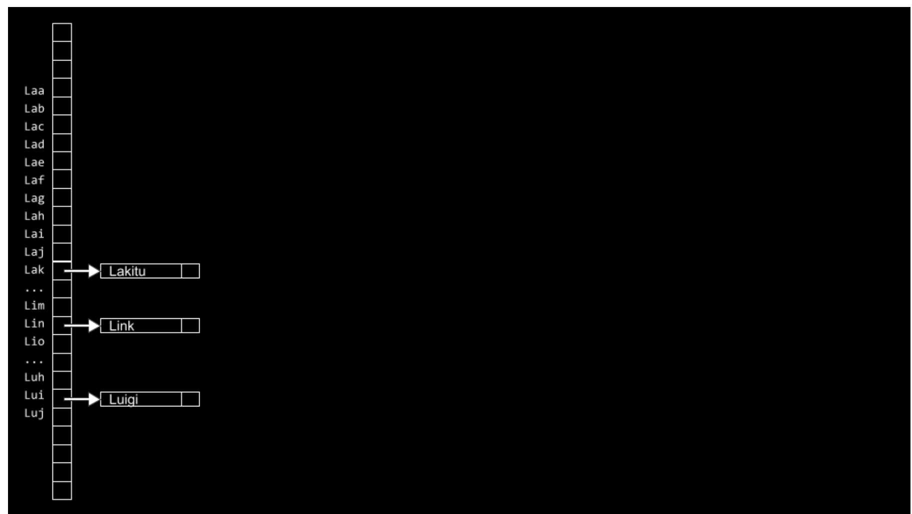
A hash table could be imagined as follows:

Notice that this is an array that is assigned each value of the alphabet. Then, at each location of the array, a linked list is used to track each value being stored there:

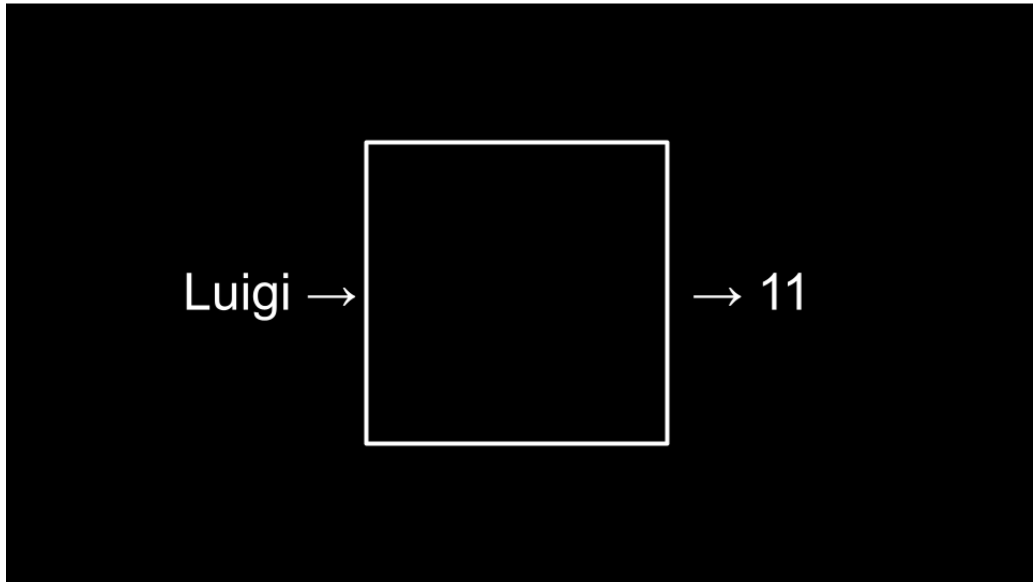


Collisions occur when you add values to the hash table, and something already exists at the hashed location. In the above, collisions are simply appended to the end of the list.

Collisions can be reduced by better programming your hash table and hash algorithm. You can imagine an improvement upon the above as follows:



Consider the following example of a hash algorithm:



This could be implemented in code as follows:

```
#include <ctype.h>

unsigned int hash(const char *word)
{
    return toupper(word[0]) - 'A';
}
```

This function hashes a word based on its first letter.

toupper(word[0]) converts the first character to uppercase.

Subtracting 'A' turns letters 'A'–'Z' into indices 0–25.

For example:

"apple" → 'A' → index 0

"Berry" → 'B' → index 1

Time Complexity of a hash table:

- Average case: O(1): very fast if items are evenly distributed.
- Worst case: O(n): happens if many items collide in the same bucket.
- Proper hash functions and sufficient table size minimize collisions and maintain near O(1) search time.

Real-world examples:

- Programming languages: Many languages, like Python, Java, and C++, use hashing for data structures like dictionaries, sets, and hash maps, to make data lookup very fast.

- Passwords: Websites don't store your actual passwords. They store hashed versions of them. This means even if someone gets access to the database, they can't see your real password.

- Databases: Hashing helps in indexing data, so databases can find records much faster, which improves performance when handling large amounts of information.

Case Study: Amazon

Amazon's shopping cart uses a hash table with 1 M slots to check item availability in < 5 ms.

Collisions? Linked-list per slot, still resulting in O(1) average.

Conclusion

Data structures form the foundation of efficient computing. From simple arrays that store data in order, to linked lists, stacks, and queues that manage how data flows, each structure serves a specific purpose. Trees and graphs allow us to represent relationships and hierarchies, while hash tables provide lightning-fast data access through hashing.

Together, these structures show that there is no one-size-fits-all solution, and the best data structure depends on the problem you’re trying to solve. Mastering them not only improves how you write code but also how you think about organizing, connecting, and optimizing information in the real world.

MASTERING CONNECTIONIST AI

A practical playbook

VOL.3

2025

SENTIENT CORE VENTURES

PRESENTED BY

CAINIAO

INTRODUCTION TO CONNECTIONIST AI

In this chapter:

- What AI is and why it's rapidly growing
- Key characteristics of Connectionist systems
- Simple analogies to understand neurons, weights, and biases

Introduction

Artificial Intelligence (AI) is a technology that enables computers and machines to do work that normally needs a human to operate. During the last few decades, the growth of AI has exponentially risen. The reason for that is the availability of vast amounts of data for AI to learn from, as well as due to the increase in power and efficiency of microprocessors.

Connectionist AI, commonly associated with neural networks and deep learning, is inspired by the structure of the human brain. It doesn't rely on rules and symbols to solve problems (like with symbolic AI); instead, it gains the knowledge of how to solve the problem through vast amounts of data and learning.

Key Characteristics:

- Distributed Representation: knowledge is encoded in many units (neurons for neural networks) and is not explicitly represented.
- Learns from Data: connectionist models learn patterns from data such as images and texts, through a process of training.
- Bottom-up approach: intelligence is not programmed but comes from the interaction of simple processing units

- No explicit rules: the system doesn't rely on human-defined rules or logic but adapts based on patterns in the data.

Learning Suggestions:

- **Start with analogies:** Think of neurons like light switches — weights = dimmers, bias = threshold.

- **Train a toy model:** Use Teachable Machine

But this AI software isn't REPLACING you, it's going to be MANAGING you.

HOW DOES AN AI LEARN?

In this chapter:

- The 3 main learning types: supervised, unsupervised, reinforcement
- Key algorithms like regression, classification, and clustering
- Real-world case studies (e.g., Netflix recommendations, Tesla Autopilot)

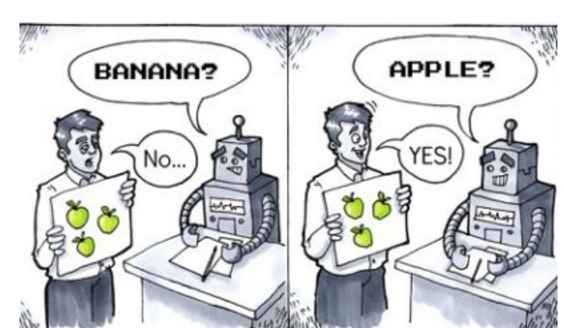
How Does an AI Learn?

There are 3 main ways that AI could learn:

1. Supervised Learning

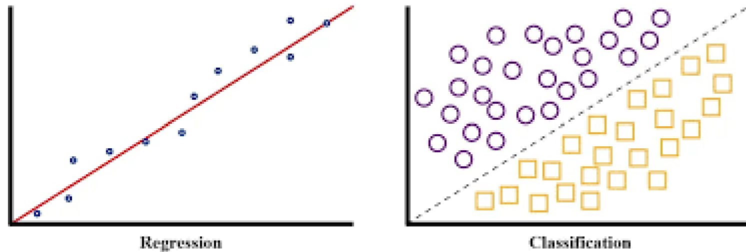
Supervised learning is a type of machine learning where an AI model learns with training labels. Meaning when the machine outputs something, there will be a human who tells the machine if it is right or wrong. Similar to how a teacher points out mistakes a student makes. The goal is for the model to learn the relationship between inputs and outputs so it can accurately predict the output for new, unseen data.

This may take a lot of time and power because of the amount of data the machine needs to accurately predict the outcome. When a machine is trained within a pool of data and then tested on with new data, it may predict some outcomes wrong because it has not learned those unfamiliar data yet and has to update the data. More inputs will make the machine predict much more accurately; however, it will consume much more power.



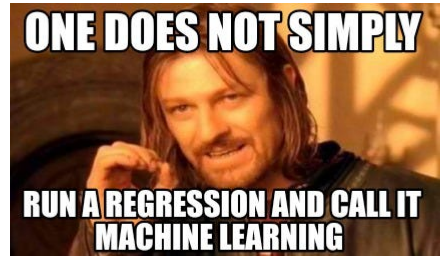
Supervised Learning

Types of Supervised Learning



a. Regression

Regression is a type of machine learning where algorithms learn from the data to predict continuous values such as sales, salary, weight, or temperature. This model learns the relationship between input and output variables to predict the outcome.



b. Classification

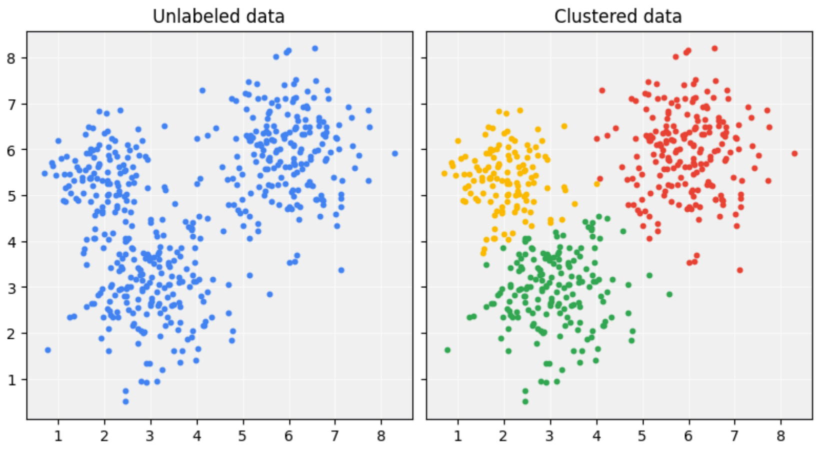
This learning system will categorize the inputs into their own groups. Classification algorithms are used for predicting discrete outcomes; these values can be anything, such as True/False, Yes/No, and others. An example of this would be email spam detection (spam or not spam).

Unsupervised learning is a type of machine learning where the model learns without training labels. The goal is to find hidden patterns or relationships within the data without guidance on what the correct answers should be.

2. Unsupervised Learning

Types of Unsupervised Learning

1. Clustering



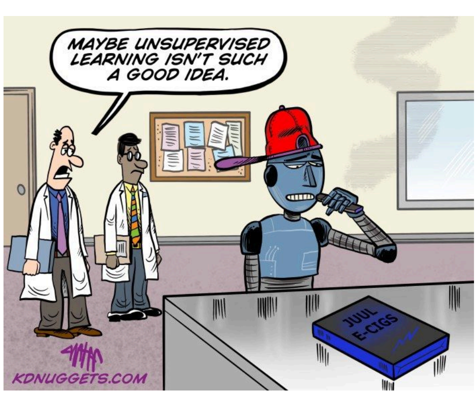
a. Groups similar data points based on feature similarity.

b. Goal: Discover natural groupings in the data.

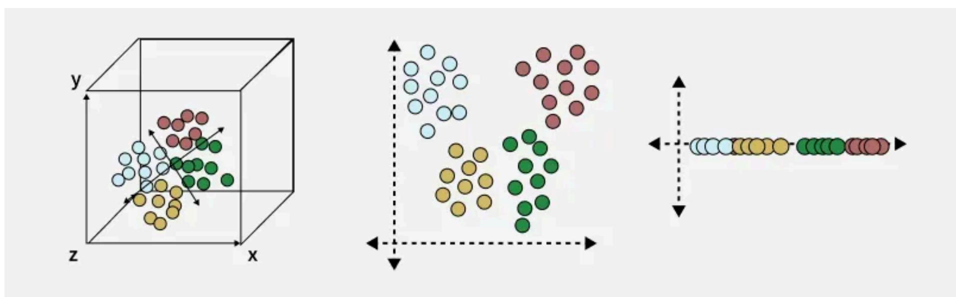
c. Examples:

- Customer segmentation in marketing
- Document or topic grouping

d. Algorithms: K-Means, Hierarchical Clustering, DBSCAN.



2. Dimensionality Reduction



a. Simplifies data by reducing the number of features while keeping important information.

b. Goal: Make data easier to visualize or process.

c. Examples:

- Visualizing complex datasets
- Reducing noise in data

d. Algorithms: Principal Component Analysis (PCA), t-SNE, Autoencoders.

Case Study: Netflix Recommendations

Problem: Suggest shows you'll actually binge without asking you any questions.

Concept Used: Clustering (k-means + matrix factorization) on viewing history.

Result: 80% of watched content comes from recommendations.

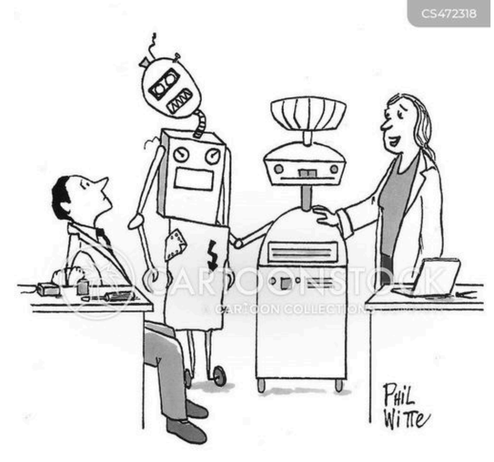
Lesson: Unsupervised learning finds hidden patterns in messy human behavior.

3. Reinforcement Learning

In reinforcement learning, the machine does trial and error to find a solution to the problem. Depending on what it does, it receives reward or punishment through its actions. The goal is to maximize the reward it gets. Unlike supervised learning, the machine has to figure out how to solve the problem by itself through trial and error. It is similar to how a baby learns to adapt to the world. When a baby eats dirt, the unpleasant flavor will create a negative signal, making it avoid eating dirt or similar things in the future. However, when it eats something delicious, it perceives it as a reward, reinforcing it to seek out similar foods again.

Types of reinforcement:

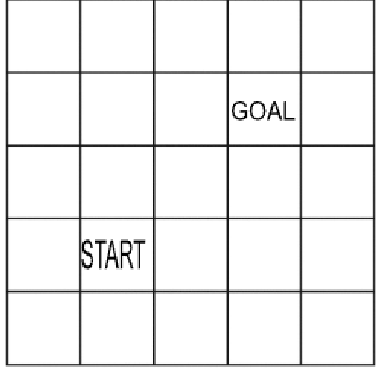
- **Positive:** When a positive reinforcement happens, the frequency of the behaviour will increase.
- **Negative:** When a negative reinforcement happens, the negative action will be avoided.



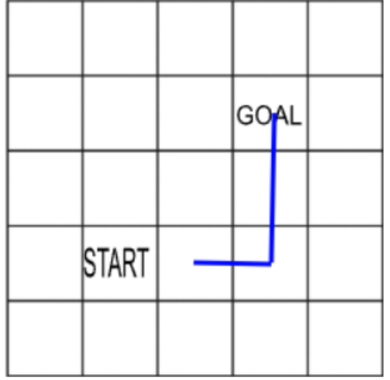
CARTOONSTOCK
"This robot learns from its mistakes--except when it comes to personal relationships."

Credit Assignment

Credit assignments are needed to find the most optimal solution to a problem, by assigning a value to each action it takes. For example, let's have a 5x5 grid with a starting point and a goal.



Now, the goal of the machine is to find a path to the goal from the starting point. Let's say the machine took this approach:



Since the machine has already figured out a path on how to get to the goal, it can always just repeat the same path to get to the goal. However, this may not be the most efficient path to reach the goal. To recognize whether or not the path is the most efficient or not, it will explore other paths until it has covered every grid and assign each of them a value, with the grid near the goal as the highest value. Now, the machine will know the most efficient path to reach the goal by following those high-value grids. It may look something like this:

0.20	0.25	0.33	0.5	0.33
0.25	0.33	0.5	GOAL	0.5
0.20	0.25	0.33	0.5	0.33
0.17	START	0.25	0.33	0.25
0.14	0.17	0.20	0.25	0.20

Case Study: Tesla Autopilot

Problem: Teach a car to change lanes safely without a human writing 1 million rules.

Concept Used: Deep Q-Networks (DQN), which is a type of reinforcement learning.

Result: Tesla cars auto-lane-change millions of times per day — and crash rates drop.

Lesson: RL works when you can simulate millions of trials (and tolerate some mistakes).

CHAPTER

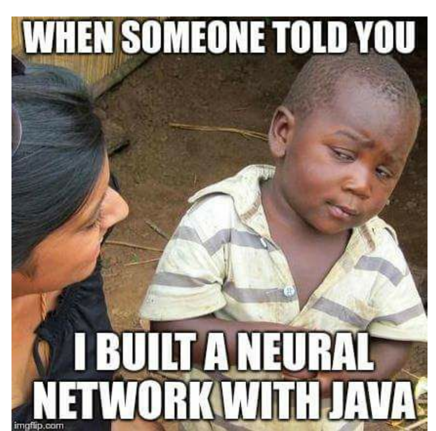
NEURAL NETWORKS

In this chapter:

- What perceptrons and artificial neural networks are
- The structure of neural networks: input, hidden, output layers
- How training works: forward pass, loss functions, backpropagation

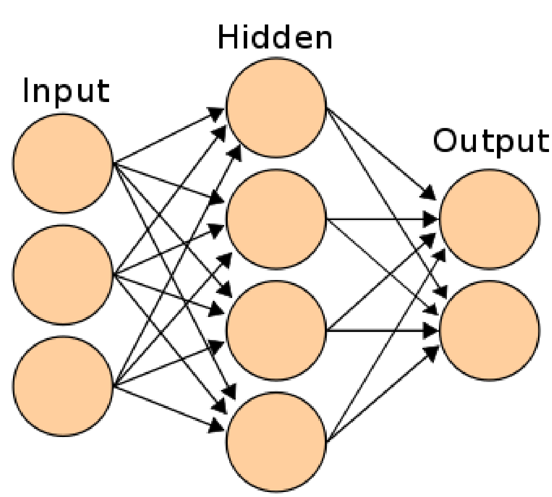
Neural Networks

A Neural Network is a type of AI model inspired by how the human brain works. It's made up of layers of connected nodes (neurons) that process information and learn patterns from data. The most simplest type of neural network is called a perceptron, which only consists of 1 single artificial neuron. If we connect multiple perceptrons together, we can create what is called an Artificial Neural Network (ANN).



Structure of ANN:

All neural networks are made of 3 main layers,



1. **Input Layer:** This is where the neural network receives data represented as numbers. Each input neuron represents a single feature of the data. For example, a pixel value of an image, word frequencies, age, or anything that can be converted to a number.

2. **Hidden Layer**: This layer, between the input and the output layer, is where the input will be processed. These layers could consist of multiple hidden layers, depending on how complex the problems are. They will perform computations on the input data by applying **weights** and **biases** to the inputs, then processing them through an **activation function.**

3. **Output Layer:** This is where the network produces its prediction. It produces the network's output, by a number which represents how sure the machine's predictions are. The number of neurons in the output layer corresponds to the number of possible outcomes or classes.

THIS IS A NEURAL NETWORK.

IT MAKES MISTAKES.
IT LEARNS FROM THEM.

BE LIKE A NEURAL NETWORK.

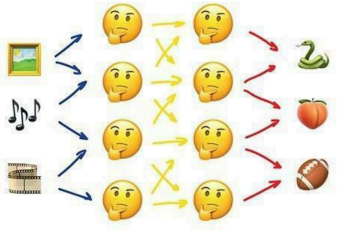
input neurons
hidden neurons
output neurons
@redcoders

Weight

Weights are parameters that control the strength of the connection lines between the neurons from one layer to the next. The value of the weight

can vary to amplify (positive weight), reduce (negative weight), or even nullify (zero weight) the output signal.

Bias



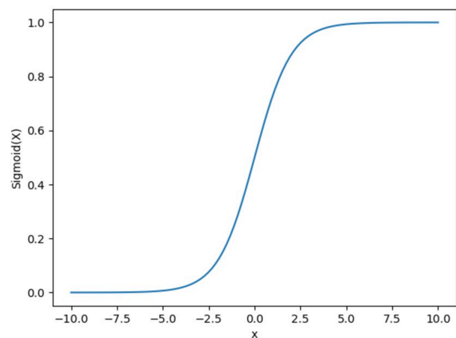
Bias are parameters added to the weighted sum of the inputs to adjust the output. Basically, bias is a number that tells how high the weighted sum needs to be before going through the activation function. Biases are not connected to the inputs but are added to the output.

Activation function

An activation function is a mathematical function that determines whether or not a neuron should be activated by further calculating the weighted sum and bias. Here are some popular Activation functions:

1. Sigmoid Function

The output using this function will represent probabilities which compress the relevant weighted sum into an interval of 0 to 1.



$$\sigma(x) = \frac{1}{1+e^{-x}}$$

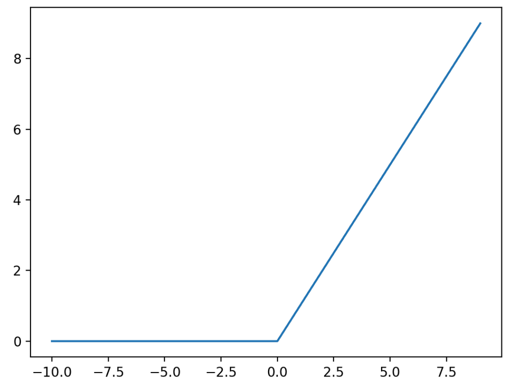
x = weighted sum inputs of the neuron

e = euler’s number ≈ 2,718

- For very negative values of x, the output of the sigmoid is close to 0.
- For large positive values of x, the output is close to 1.
- Around x = 0, the sigmoid function gives an output of 0.5. The curve will transition steeply within this area.

2. ReLU (Rectified Linear Unit) Function

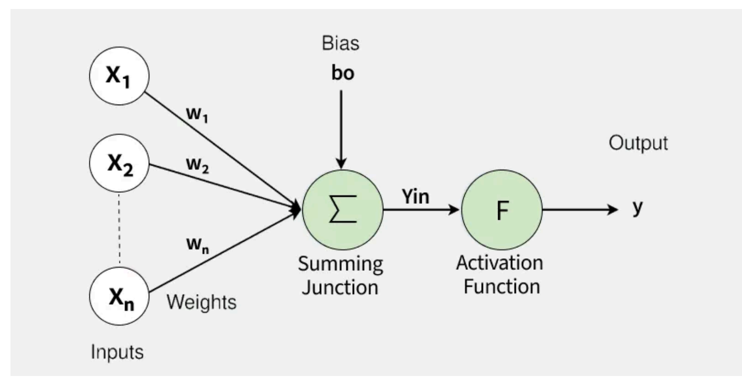
This function decides whether a neuron should be activated or not by checking if the input is positive or negative.



$f(a) = max(0, a)$

- If a > 0, then f(a) = a
- If a < 0, then f(a) = 0

The Formula



$$y = f(w_1x_1 + w_2x_2 + ... + w_nx_n + b)$$

x = input values

w = corresponding weights

b = bias

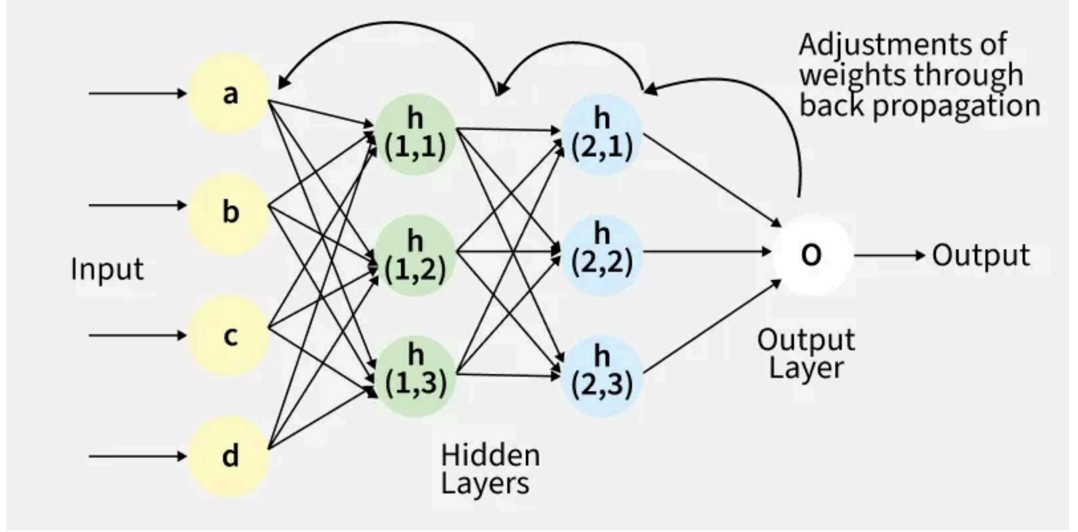
f = activation function

y = output

Training Neural Networks

Training a neural network means teaching it to make accurate predictions by adjusting its parameters (weights and biases) based on data. The training process includes:

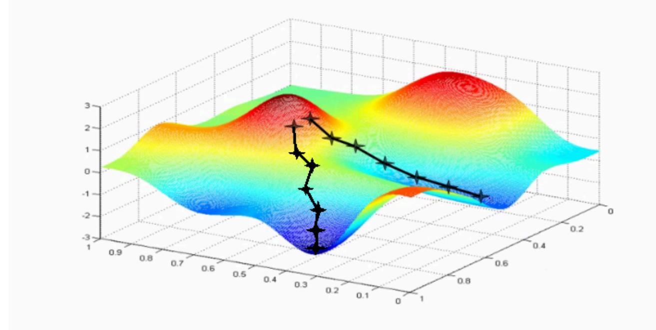
1. **Input Data**: You feed training data (images, text, numbers) into the network.
2. **Forward propagation:** This is where the input gets calculated into the weighted sum and bias, and then passes through an activation function, which becomes the output for the next layer.
3. **Loss function:** The difference between the predicted answer and the correct answer
4. **Backpropagation:** This will look at the loss function of the predicted answer and blame it on the previous neurons to get their weights and biases adjusted.



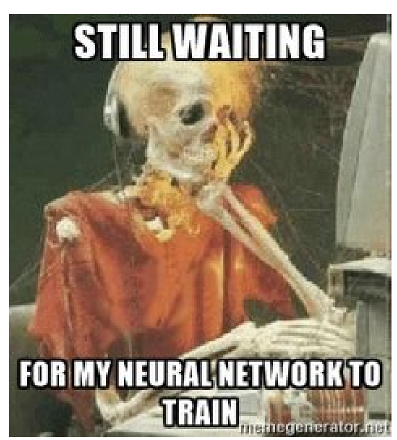
5. **Gradient Descent:** Is a function which is used to minimize the loss function. Providing both the magnitude and the direction of the loss functions shows us the direction of adjustment. To calculate this, we have to take the derivative of the loss function and calculate.

$$w_{new} = w_{old} - \frac{\eta \partial L}{\eta \partial w}$$

- L = loss function
- $\frac{\eta \partial L}{\eta \partial w}$ = gradient (slope of the loss function with respect to the weight)
- η = Learning rate (how big each step is)



We can think of it like a mountain. The loss function is a 3D mountain surface, where each point represents a set of weights and its error (height). The goal is to find the lowest point (the minimum loss). You look at the slope of the mountain and step downhill, repeating until you reach the bottom. The machine might find the lowest point of a valley and stop, but we don't know whether there is another valley with a deeper bottom. This is called a **Local Optimal Solution**. To find the **Global Optimal Solution**, we can repeat the steps, but using other starting points or have multiple processors that start from different points.



6. **Repeat:** Repeat the steps until its predictions are stable

- Epoch = one full pass through all training data.
- Batch = subset of data processed before weights are updated.
- Mini-batch training = uses small chunks (32 or 64 samples per batch) for efficiency.

POPULAR NEURAL NETWORKS

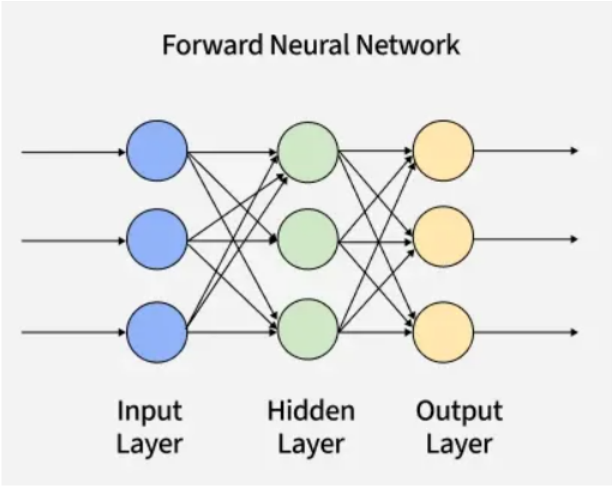
In this chapter:

- Popular Neural Networks
- Advantages, disadvantages, and real-world applications of each
- When to use FNNs, RNNs, CNNs, LSTMs, GRUs, or RBFNs

Popular Neural Networks

Feedforward Neural Network (FNN)

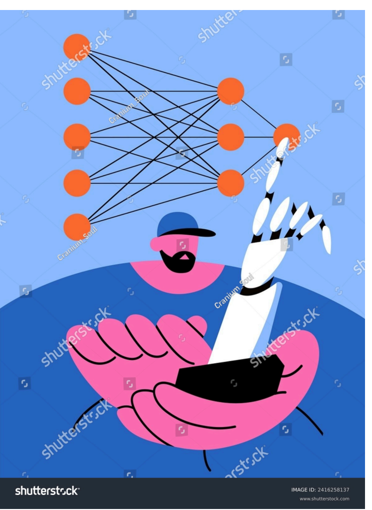
A Feedforward Neural Network (FNN) is a kind of artificial neural network in which the connections between nodes do not create any loops during the forward pass. It is called "forward" because the data travels in a single direction—from the input layer to the hidden layers and finally to the output layer—without any backward or looping connections.



The structure of FNNs ensures that each layer connects only to the next one, with no backward loops. Although backpropagation relies on previous errors to adjust weights, it doesn't change how the network is structured during prediction.

Advantages of FNN

- **Simple Structure:** It's easy to understand and implement.
- **Universal Approximation:** With enough hidden layers and neurons, FNNs can model almost any function, making them adaptable for many uses
- **No Cycles:** Since data moves in only one direction, both training and implementation are straightforward.



Disadvantages of FNN

- **Limited Memory:** FNNs cannot remember past inputs or capture time-based patterns, making them unsuitable for sequence data.
- **Scalability:** For complex problems, FNNs may require numerous layers and neurons, increasing computational cost.

Applications of FNN

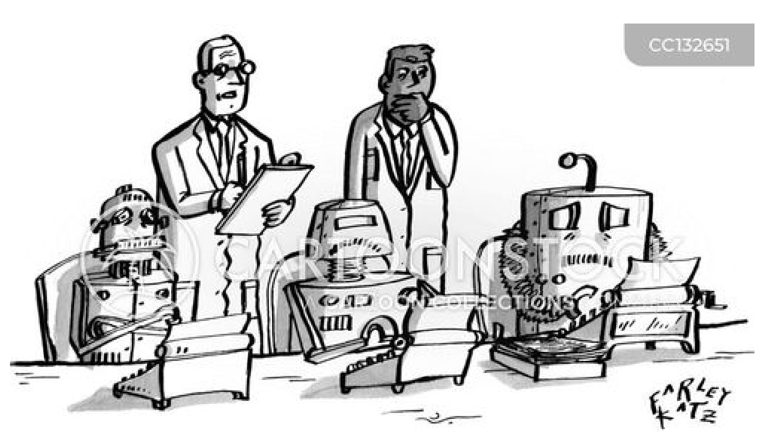
- **Image Classification:** Identifying objects, faces, or handwriting in images.

- **Natural Language Processing:** Text classification (e.g., spam detection, sentiment analysis).
- **Financial Forecasting:** Predicting stock prices or market trends.

Medical Diagnosis: Classifying diseases or detecting anomalies in medical scans.

Recurrent Neural Network (RNN)

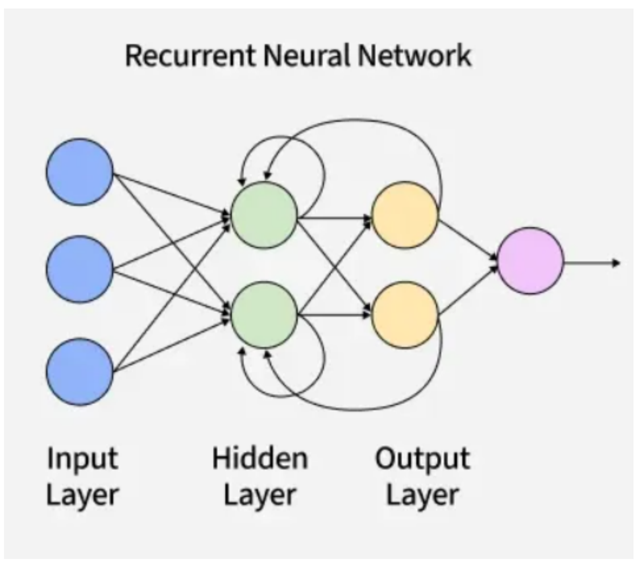
A Recurrent Neural Network (RNN) is a type of neural network where the output from one step is used as input for the next step. Unlike traditional neural networks, where inputs and outputs are treated as independent, RNNs are designed to handle sequential data, making them ideal for tasks where order and context matter, such as time series forecasting, language processing, and speech recognition.



"The robots have become self-aware and self-loathing. Now all they do is write novels."

Key Concepts of RNNs

1. **Sequential Data Processing:** RNNs can handle sequences like sentences or time series because they store past information in an internal memory, unlike feedforward networks.
2. **Recurrent Connections:** They have loops that let information persist between steps. The output from one stage becomes the input for the next, helping the model retain context.
3. **Shared Weights:** The same weights are reused at each time step, reducing parameters and improving efficiency for sequence-based tasks.



RNNs update their internal state at every time step, allowing them to remember previous inputs. Each neuron receives input from both the current data and the previous output, creating a feedback loop. This structure enables RNNs to predict or classify based on past context.

Example:

In natural language processing, if the input is *"The cat is"*, the RNN may predict *"sleeping"* as the next word because it understands the preceding context.

Types of RNNs

- **One-to-One:** Standard model where one input gives one output.

  Example: Image classification — an image is input, and the network outputs a label like "cat" or "dog."

- **One-to-Many:** One input produces multiple outputs over time.

  Example: Image captioning — an image generates a sequence of words describing it.

- **Many-to-One:** A sequence of inputs gives a single output.

  Example: Sentiment analysis — input a sentence, output a sentiment label like "positive" or "negative."

- **Many-to-Many:** Both inputs and outputs are sequences.

  Example: Machine translation — a sentence in one language is translated into another.

Applications of RNN in real life

- **Language Modeling & Text Generation:** Generate text, predict next words, and build chatbots or autocomplete tools.
- **Sentiment Analysis:** Analyze text to determine emotional tone or opinion.
- **Stock Market Prediction:** Forecast stock movements by analyzing historical financial data.
- **Patient Monitoring:** Track patient vitals like ECG or EEG to detect anomalies.
- **Intrusion Detection:** Monitor network activity logs to identify unusual or malicious patterns.

Long Short-Term Memory (LSTM)

Long Short-Term Memory (LSTM) networks are a specialized form of Recurrent Neural Networks (RNNs) designed to retain information over long periods. They use memory cells to store both short-term and long-term information, making them ideal for handling sequences where context matters across time. What sets LSTMs apart is their gating mechanisms, which control how information is stored, updated, or forgotten. LSTMs are widely used in language translation, speech recognition, and other tasks requiring long-term context understanding.

Key Components of LSTM

Cell State:

Serves as the network’s long-term memory, carrying information through different time steps.

Forget Gate:

Determines which parts of the cell state to discard. It produces values between 0 and 1 — 0 means “forget completely,” and 1 means “keep completely.”

Input Gate:

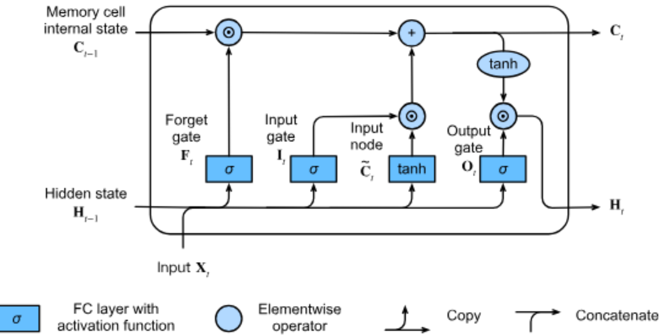
Controls which new information gets stored in the cell state. It includes a sigmoid layer (to choose which values to update) and a tanh layer (to create new candidate values).

Output Gate:

Filters the cell state to decide what information should be output. The result is a processed version of the memory.

Cell Update:

Merges the old cell state with the new candidate information to update the memory.



Advantages of LSTM

- **Capturing Long-Term Dependencies:** LSTMs effectively manage information flow through input, forget, and output gates, allowing them to remember important patterns across long sequences.
- **Reducing the Vanishing Gradient Problem:** Unlike standard RNNs, LSTMs maintain more stable gradients during training, making it easier to learn long-term relationships.
- **Selective Memory Control:** Their gate-based design allows selective remembering or forgetting, providing more flexibility in managing sequence information.

Disadvantages of LSTM

- **High Computational Demand:** LSTMs are resource-intensive due to their multiple gates, increasing both memory usage and processing time.

- **Slow Training:** Their complex structure requires longer training periods, especially on large datasets.
- **Limited Parallelization:** Since LSTMs process data sequentially, they can’t fully utilize parallel computation, slowing down training compared to architectures like CNNs.
- **Hyperparameter Sensitivity:** LSTMs require fine-tuning of parameters such as learning rate and dropout rate, making optimization more challenging and time-consuming.

Gated Recurrent Unit (GRU)

Gated Recurrent Unit (GRU) is a streamlined version of the Long Short-Term Memory (LSTM) network that merges the forget and input gates into a single gate. This simplification reduces computational demands while maintaining much of the LSTM’s performance. GRUs are faster to train, require fewer parameters, and often achieve comparable results to LSTMs. They are particularly useful in real-time applications like speech synthesis and video prediction, where speed and efficiency are essential.

Key Components of GRU

1. **Update Gate:**

Controls how much past information should be carried forward. It serves as a combination of the forget and input gates found in LSTMs.

2. **Reset Gate:**

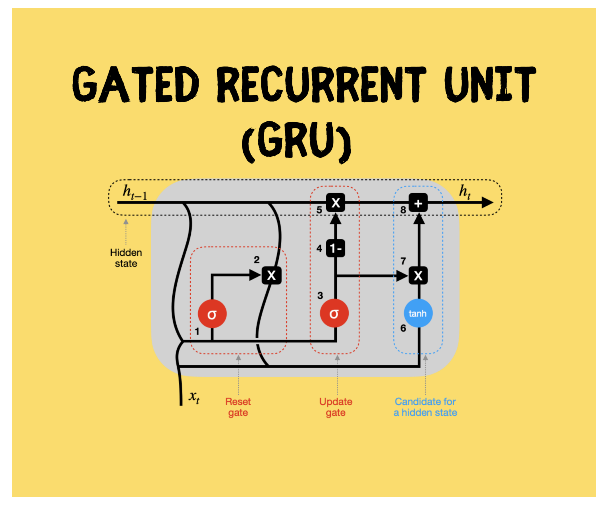
Determines how much of the previous hidden state should be forgotten, influencing how new information is integrated.

3. **New Memory Content:**

Uses the reset gate to create new candidate information for memory by filtering parts of the previous hidden state.

4. **Final Memory:**

The updated hidden state blends the old state with the new candidate information, maintaining relevant data for the next step.



Differences Between LSTM and GRU

- **Complexity:**

LSTMs use three gates (forget, input, and output) plus a separate cell state, while GRUs have only two gates (update and reset) and no distinct cell state, making them simpler and faster.

- **Performance:**

LSTMs excel in handling long-term dependencies, whereas GRUs are more efficient and perform similarly on smaller datasets or shorter sequences.

- **Memory Usage:**

LSTMs consume more memory due to their additional components, while GRUs are lighter and better suited for devices with limited resources.

- **Use Cases:**

LSTMs are ideal for tasks like language translation or speech recognition, where remembering long-term context is essential. GRUs work best in real-time or resource-constrained scenarios where computational efficiency is key.

Advantages of GRU

- **Simplified Architecture:**

By merging gates, GRUs reduce parameter count, making them easier and faster to train than LSTMs.

- **Computational Efficiency:**

Fewer gates mean faster training and lower memory usage, which is beneficial for mobile or embedded systems.

- **Reduced Overfitting:**

With fewer parameters, GRUs are less prone to overfitting, especially on smaller datasets.

- **Ability to Capture Long-Term Dependencies:**

Despite their simplicity, GRUs effectively handle long-term relationships in sequential data.

Disadvantages of GRU

- **Less Control Over Memory:**

Having fewer gates limits the model’s fine-tuning ability for handling complex temporal dependencies.

- **Hyperparameter Sensitivity:**

GRUs require careful tuning of training parameters, and the absence of separate forget and input gates can complicate optimization.

- **Not Always Ideal for Very Long Sequences:**

LSTMs can outperform GRUs in tasks involving extremely long-term dependencies.

- **Lower Industry Adoption:**

  LSTMs remain more popular and well-studied, so GRUs may have fewer dedicated tools and research resources in certain domains.

Convolutional Neural Network (CNN)

A Convolutional Neural Network (CNN) is a deep learning architecture designed to efficiently process data with a grid-like structure, such as images. CNNs are especially powerful in image-related tasks because they can automatically detect spatial features—starting from simple edges to complex shapes—through the use of convolutional layers. They are widely used in object detection, facial recognition, and medical imaging analysis due to their ability to extract meaningful visual patterns from raw data.

Key Components of CNN

1. **Convolutional Layer:**

   This is the fundamental layer of CNNs. It uses small filters (kernels) that slide over the input data to extract important features. Each filter captures different aspects—such as edges, corners, or textures—and deeper layers detect higher-level patterns like shapes or faces.

Example: In an image, early layers might detect horizontal or vertical lines, while deeper layers identify complex objects like eyes or cars.

2. **Activation Function (e.g., ReLU):**

After convolution, the output is passed through an activation function like **ReLU (Rectified Linear Unit)** to introduce non-linearity. This enables the model to learn complex patterns and relationships beyond simple linear transformations.

3. **Max Pooling:**

Max pooling reduces the spatial size of the feature map while retaining the most important features. It selects the maximum value within a small window (e.g., 2×2) and moves this window across the feature map.

Benefits:

- Keeps dominant features such as edges or textures.
- Adds **translation invariance**, meaning small shifts in the input won't significantly change the output.

4. **Average Pooling:**

Instead of selecting the maximum value, average pooling calculates the average of values within each window.

Benefits:

- Produces smoother feature maps and reduces noise.
- Useful when maintaining background or texture information is important.

5. **Fully Connected Layer:**

After convolution and pooling, the data is flattened into a 1D vector and passed through one or more fully connected layers. These layers combine the extracted features to make the final prediction.

Example: In image classification, the final fully connected layer outputs probabilities for each class using the **softmax** activation function.

6. **Output Layer:**

The final layer provides the model's predictions—often as class probabilities for classification tasks or as continuous values for regression problems.

Advantages of CNN

- **Automatic Feature Extraction:**

CNNs automatically learn useful features from raw data, removing the need for manual feature engineering.

- **Spatial Invariance:**

Through convolution and pooling, CNNs learn to recognize objects regardless of their position within the image.

- **Parameter Efficiency:**

Shared weights in convolutional layers reduce the number of parameters, making CNNs more efficient and less prone to overfitting.

- **Wide Applicability:**

CNNs excel not just in computer vision but also in video analysis, NLP, medical diagnostics, and autonomous systems.

- **Handles High-Dimensional Data:**

CNNs are specifically built for processing complex, multi-dimensional inputs like images and videos.

Disadvantages of CNN

- **High Computational Demand:**

Training deep CNNs on large datasets is resource-intensive and often requires GPUs for faster computation.

- **Large Data Requirement:**

CNNs perform best with large datasets. Limited data can cause overfitting and poor generalization.

- **Lack of Interpretability:**

CNNs act as “black boxes,” making it hard to understand how they arrive at specific predictions—a drawback in sensitive fields like healthcare.

- **Sensitivity to Rotation and Orientation:**

CNNs may misclassify rotated or differently oriented versions of the same object, though this can be mitigated with data augmentation or specialized architectures.

- **Vulnerability to Adversarial Attacks:**

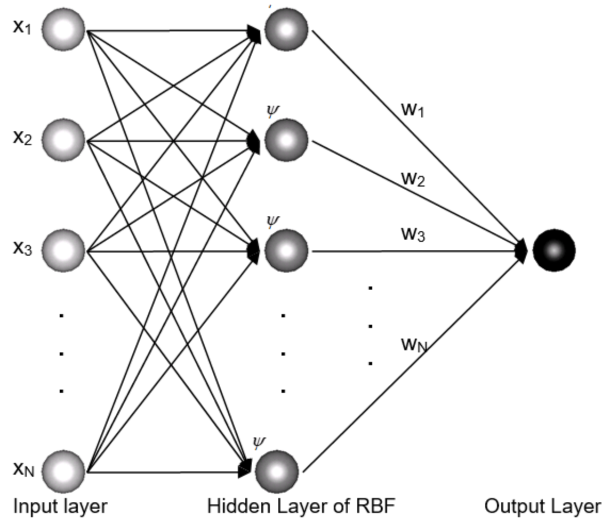
Slight, imperceptible changes in input data can fool CNNs, which poses security risks in areas like autonomous vehicles and facial authentication systems.

Radial Basis Function Network (RBFN)

A Radial Basis Function Network (RBFN) is a type of artificial neural network that uses radial basis functions as activation functions. It's typically used for regression, classification, and time series prediction tasks.

Key Components

1. **Localized Response:** RBFs are highly responsive when the input is near a specific center and gradually less responsive as it moves farther away.
2. **Radial Symmetry:** The output is determined solely by the distance from the center, not the direction—hence the term *radial*.
3. **Non-linear Activation:** These functions introduce non-linearity, enabling the network to capture complex data relationships.
4. **Training Process:** Training usually occurs in two steps—first, selecting the function centers (commonly using *k-means clustering*), and then adjusting the output weights using linear methods such as *least squares*.



Why Use RBFNs?

Radial Basis Function Networks are especially powerful in the following scenarios:

- **Function Approximation:** They can model complex, non-linear functions by producing localized responses for different input regions, ideal for regression problems requiring smooth mappings between inputs and outputs.
- **Classification:** RBFNs work well when data is clustered, as they can assign centers near each class and classify new points based on proximity.
- **Simplicity:** Compared to deep networks, RBFNs have fewer parameters, train faster, and are easier to implement while still producing effective results.

Advantages of RBFNs

- **Fast and Simple Training:** They train faster than multilayer perceptrons (MLPs) since training is divided into two stages—choosing centers (via clustering) and fitting output weights (via linear regression).

  Localized Activation: Each neuron reacts strongly to nearby data points, making RBFNs effective for highly non-linear relationships.
- **Strong Function Approximation:** They can smoothly approximate complicated functions, especially when interpolation is needed between known points.
- **Noise Tolerance:** The distance-based activation makes them less sensitive to minor variations or noise in the input data.
- **Interpretability:** Each neuron represents a specific region in the input space, making it easier to understand its role in producing the final output.

Disadvantages of RBFNs

- **Poor Scalability with High Dimensions:** RBFNs become computationally expensive as data dimensionality increases, requiring many neurons to cover the input space effectively.
- **Dependence on Center Selection:** Network performance relies heavily on how well the centers are chosen; poorly selected centers can lead to weak generalization.

- **Parameter Sensitivity:** The spread (radius) parameter must be carefully tuned—too small leads to overfitting, while too large causes underfitting and poor data representation.

KEY APPLICATIONS OF CONNECTIONIST AI

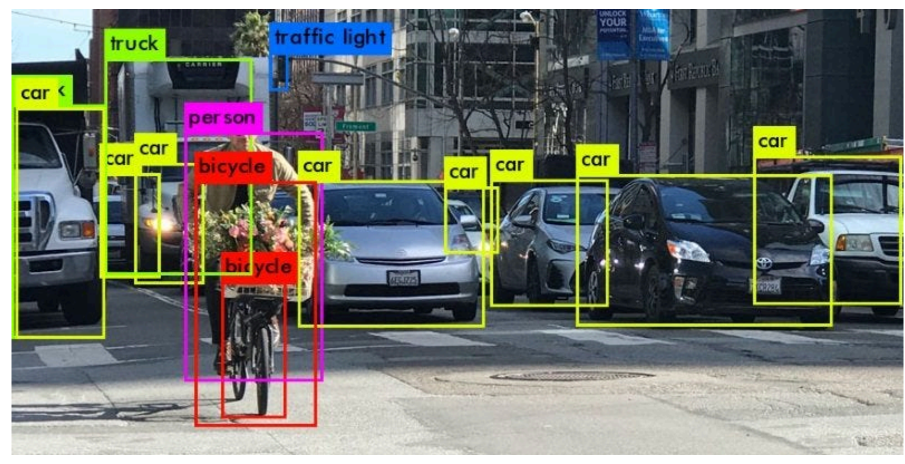
In this chapter:

- Real-life applications of Connectionist AI

Key Applications of Connectionist AI

Image Recognition

Connectionist AI, particularly through Convolutional Neural Networks (CNNs), is widely used in image classification and object detection. These systems can automatically identify and label objects, faces, or scenes in images, making them essential for technologies such as facial recognition, medical image analysis, and autonomous vehicles.



Speech Recognition

Neural networks power modern speech-to-text systems by learning to map audio signals to words and phrases. This allows for natural and accurate voice interfaces used in virtual assistants, transcription services, and automated customer support systems.

Autonomous Driving

Connectionist AI enables vehicles to perceive and interpret their surroundings through camera and sensor data. Neural networks help detect lanes, pedestrians, and obstacles, contributing to navigation and real-time decision-making.

Medical Diagnosis

Neural networks assist doctors by analyzing medical data such as X-rays, MRIs, and patient records. They can detect diseases, classify medical images, and predict potential health risks with high accuracy.

Time Series Prediction

Networks like RNNs and Long Short-Term Memory (LSTM) models are used to forecast future values based on historical data. Common use cases include weather forecasting, stock market prediction, and demand estimation.

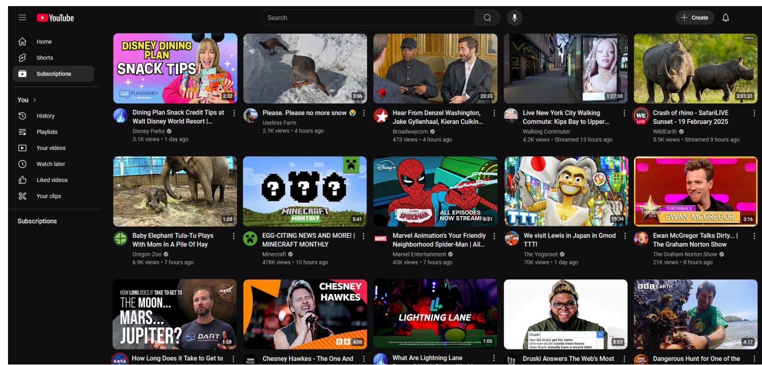
Robotics

Neural networks help robots learn and adapt to their environments by processing sensor inputs and making movement or manipulation

decisions. This enables applications in manufacturing, exploration, and service industries.

Recommendation Systems

By identifying patterns in user preferences, connectionist AI models suggest relevant movies, products, or content. They are a key component of platforms like YouTube, Netflix, and e-commerce sites.



CHALLENGES IN CONNECTIONIST AI

In this chapter:

- Why neural networks require large amounts of data
- Challenges in Connectionist AI throughout the years

Challenges in Connectionist AI

Connectionist AI has made significant strides in recent years; however, it still faces several challenges.

Data Requirements

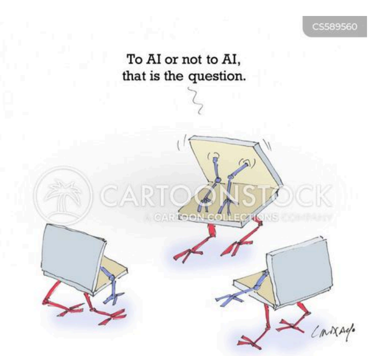
Neural networks typically require vast amounts of labeled data for training. The quality and quantity of data significantly influence the model's performance.

Example: In image classification tasks, a model like a Convolutional Neural Network (CNN) requires thousands, if not millions, of labeled images to accurately learn features that distinguish different categories. The ImageNet dataset, often used for training large models, contains over 14 million images across 20,000 categories. Acquiring such datasets can be challenging, particularly in specialized domains like medical imaging, where annotating images requires expert knowledge.

Interpretability

Neural networks are often viewed as "black boxes," making it difficult to understand how they arrive at specific decisions. This lack of interpretability can be problematic in fields where understanding the rationale behind decisions is crucial.

Example: In healthcare, a deep learning model might be used to predict patient outcomes based on medical images. If the model recommends a treatment based on its predictions, doctors need to understand why the model arrived at that conclusion. If the model suggests surgery, a clinician must know the contributing factors to ensure it aligns with the patient’s condition. Tools like LIME (Local Interpretable Model-agnostic Explanations) and SHAP (Shapley Additive exPlanations) have been developed to provide insights, but these methods are still evolving and may not fully resolve interpretability issues.



Overfitting

Overfitting occurs when a model learns the training data too well, including noise and outliers, which harms its performance on new, unseen data.

Example: A neural network trained to predict housing prices might perform exceptionally well on the training dataset but poorly on a test set if it has memorized specific details rather than learned generalizable patterns. To combat overfitting, techniques like dropout (where random

neurons are deactivated during training) and early stopping (halting training when performance on a validation set begins to decline) are employed, but finding the right balance can be tricky.

Bias and Fairness

Neural networks can learn and perpetuate biases inherent in the training data. This can lead to discriminatory outcomes, particularly in sensitive applications.

Example: In facial recognition systems, if the training data predominantly includes images of lighter-skinned individuals, the model may perform poorly when identifying individuals with darker skin tones. This issue was highlighted in studies showing that some commercial facial recognition systems misidentified people of color at significantly higher rates than white individuals. Ensuring fairness requires careful data collection and preprocessing to include diverse demographic representations and implementing bias detection tools during model evaluation.

Generalization

Ensuring that a model generalizes well to unseen data remains a significant challenge, especially when training and testing environments differ.

Example: Consider a model trained to recognize cars in images captured under bright daylight conditions. If the model is later tested on images taken at night or during inclement weather, its performance may significantly decline. This issue highlights the importance of using diverse datasets that capture various conditions during training to improve the model's ability to generalize to new environments. Techniques like data augmentation (e.g., altering brightness, flipping images) can help simulate different conditions and improve robustness.

How to mitigate these challenges in the future

In order to mitigate these challenges and the risks associated with them, we have to work on:

Data Requirements

- **Data Augmentation:** Utilize techniques such as rotation, scaling, cropping, and color adjustment to artificially increase the size and diversity of training datasets, particularly in cases where obtaining new data is difficult.
- **Transfer Learning:** Leverage pre-trained models that have been trained on large datasets and fine-tune them on smaller, domain-specific datasets. This approach can significantly reduce the amount of required labeled data while still achieving high performance.

- **Synthetic Data Generation:** Explore the use of generative models (like GANs) to create synthetic data that can complement real data, especially in fields with limited access to labeled examples.

Interpretability

- **Explainable AI (XAI) Techniques:** Implement methods like LIME and SHAP that provide insights into model decision-making. These tools can help users understand the influence of various features on predictions.
- **Model Simplification:** Use simpler models or architectures where possible, as they tend to be more interpretable. Techniques such as attention mechanisms can also highlight important features, making models easier to understand.
- **Documentation and Visualization:** Enhance transparency by documenting model training processes, decisions, and performance metrics. Visualizing model behavior and feature importance can also aid in interpretability.

Overfitting

- **Regularization Techniques:** Use methods such as L1 and L2 regularization to penalize overly complex models during training, helping to reduce overfitting.

- **Cross-Validation:** Implement k-fold cross-validation to better estimate a model's performance on unseen data, providing a more reliable assessment and helping to prevent overfitting.
- **Ensemble Methods:** Combine multiple models to improve generalization. Techniques like bagging and boosting can help create more robust predictions by aggregating the strengths of various models.

Bias and Fairness

- **Diverse Data Collection:** Ensure training datasets are representative of various demographics, geographies, and conditions to minimize bias. Actively seek out underrepresented groups during data collection.
- **Bias Detection and Mitigation:** Employ techniques to identify and assess bias in models during the development process. This includes using fairness metrics to evaluate model performance across different demographic groups and applying algorithms that can adjust for biases.

Generalization

- **Domain Adaptation Techniques:** Explore methods that help models adapt to new environments and conditions, such as domain adversarial training or using domain-invariant feature extraction.

- **Robustness Testing:** Implement rigorous testing protocols that evaluate models across various scenarios and perturbations to assess their generalization capabilities.

ETHICAL CONSIDERATIONS

In this chapter:

- Why neural networks require large amounts of data
- Challenges in Connectionist AI throughout the years

Ethical Considerations

Connectionist AI, while powerful and transformative, raises several ethical issues that must be carefully considered to ensure responsible use and development. Because these systems learn patterns from large datasets and operate as "black boxes," their decisions can sometimes be difficult to explain or control.

Bias and Fairness

Connectionist systems learn directly from data, which means that if the training data contains biases, the model will likely reproduce and even amplify them. For example, an AI trained on unbalanced datasets may favor certain groups over others in hiring, facial recognition, or loan approvals. Ensuring fairness requires careful dataset selection, continuous monitoring, and the use of bias-mitigation techniques.

Transparency and Explainability

Many neural networks, especially deep models, are difficult to interpret. Their internal reasoning process is often hidden, making it hard for humans to understand why a particular decision was made. This lack of transparency can lead to mistrust and makes accountability challenging in critical fields like healthcare, finance, or law enforcement.

Privacy and Data Security

Training Connectionist AI often requires large amounts of personal or sensitive data. Without proper safeguards, this data could be exposed or misused. Ethical AI development involves implementing strict data protection measures, obtaining informed consent, and ensuring that user data is anonymized or securely stored.

Employment and Societal Impact

Automation powered by Connectionist AI may replace certain jobs or change the nature of work. While it can improve efficiency, it also raises concerns about unemployment, inequality, and the need for reskilling workers. Ethical AI deployment involves considering its broader impact on society and promoting fair transitions.

Misuse and Security Risks

Connectionist AI can be exploited for harmful purposes, such as deepfakes, surveillance, or misinformation generation. Ethical use involves setting boundaries, implementing safeguards, and ensuring that AI technologies are developed and used for beneficial and lawful purposes.