Lesson 3

Data Structures and Algorithms
DSA

In this lesson we will talk about:

- data structures
- algorithm design
- algorithm efficiency
- searching and sorting algorithms

Data Structures

What is a data structure?

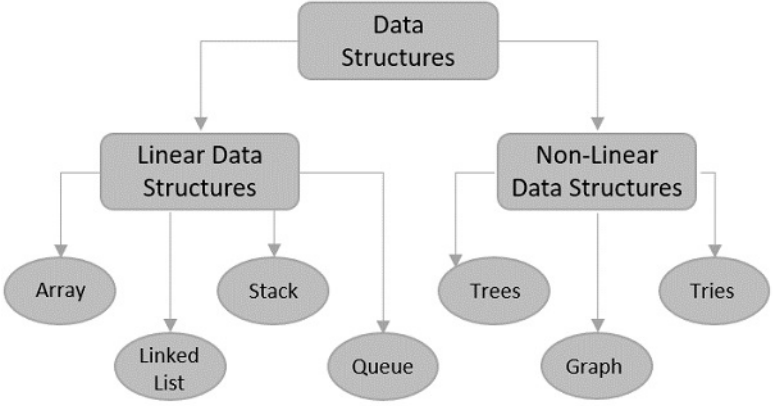
In computer science, a **data structure** is a data organization and storage format that is usually chosen for efficient access to data.[1][2][3] More precisely, a data structure is a collection of data values, the relationships among them, and the functions or operations that can be applied to the data,[4] i.e., it is an algebraic structure about data.

Data Structures

- How do we **organize** data
- For a very **efficient** access

It's a collection of data values, and the relationships among these values

Data Structures



Linear Data Structures: Arrays, Queues, Stacks

- Elements are arranged sequentially, one after the other
- The first element added will be the first one to be accessed or removed, and the last element added will be the last one to be accessed or removed
- Can have either fixed or dynamic sizes
- Offer very efficient data access

Non-linear Data Structures: Trees, Graphs

- Elements are arranged hierarchical
- We can't traverse all the elements in a single run only
- There are multiple levels which we must traverse
- It is more difficult to implement

Data structures

- Sets
- Arrays and Matrices
- Stacks
- Queues
- Linked lists
- Trees
- Graphs

Sets

Sets

A set is usually a **collection** of different things, fixed in size. Sets can also change size, usually when an algorithm will perform modifications against the set. We call these dynamic sets. These sets can change in size, grow or shrink, basically change over the time.

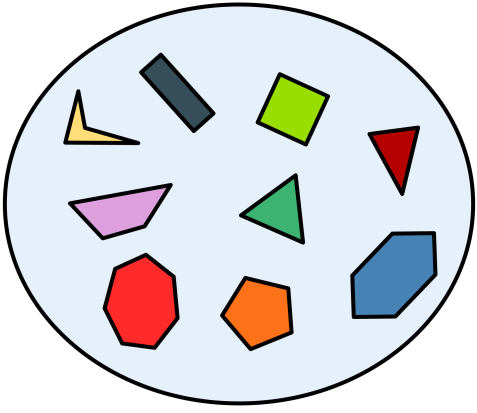
But what is a set?

Sets

The basic, fundamental data structure: $\{1,2,4,51,9\}$

- mathematical set
- unchanging, unique elements, no duplicates
- contains a fixed number of elements: finite set
- or it can contain an infinite number of elements

For example a set of polygons



Sets

A set is a **mathematical model** of a collection of different things. A set contains elements or members, which can be mathematical objects of any kind numbers, symbols, points in space, lines, other geometrical shapes, variables, or even other sets.

Sets, examples

- $\{$white, blue, red, yellow$\}$
- The empty set $\{\}$
- Natural numbers: $\mathbb{N} = \{0, 1, 2, 3, \ldots\}$
- Natural numbers except 0: $\mathbb{N}^* = \{1, 2, 3, \ldots\}$
- Integers: $\mathbb{Z} = \{\ldots, -3, -2, -1, 0, 1, 2, 3, \ldots\}$
- Positive integers: $\mathbb{Z}_+ = \{0, 1, 2, 3, \ldots\}$

$\{\mathbf{1,2,3,4}\}$

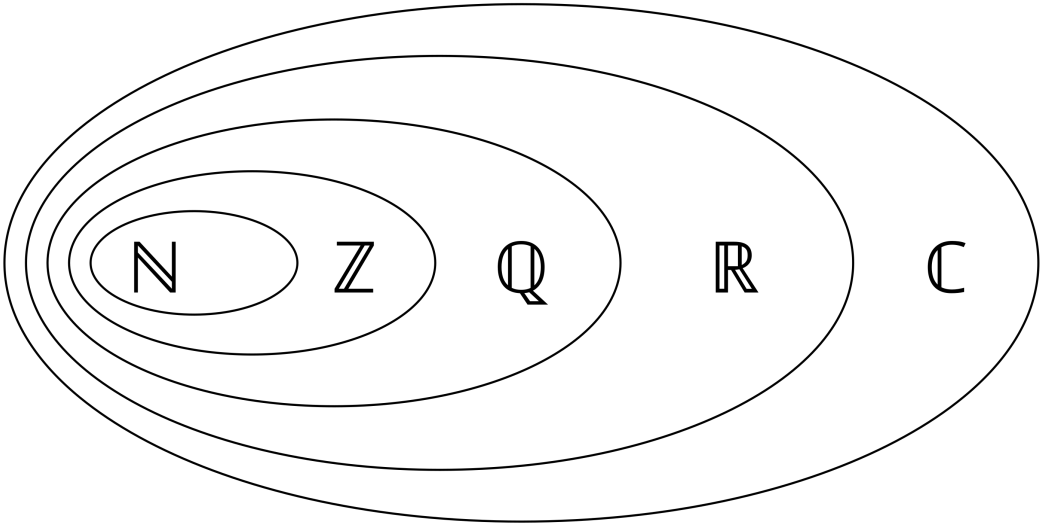
Defines a list of elements, using a simple enumeration notation (Roster notation) between curly brackets, separated by commas.

Basic Operations on Sets

- Insert - add a new element to a set
- Delete - remove an element from a set
- Test - if element X belongs to a set or not

A dynamic set which supports all these basic operations: **a dictionary**

Natural Numbers (Counting Numbers) ($\mathbb{N}$)	Numbers you use for counting: 1, 2, 3 ...	It's "natural" to count on your fingers: 1, 2, 3, ...
Whole Numbers	The natural numbers, plus 0: 0, 1, 2, 3 ...	The word "whole" has an "o" in it, so include 0.
Integers ($\mathbb{Z}$)	Whole numbers, their opposites (negatives), plus 0: ... −2, −1, 0, 1, 2 ...	Integers can be separated into negative, 0, and positive numbers.
Rationals ($\mathbb{Q}$)	Integers and all fractions, positive and negative, formed from integers. These include repeating fractions, such as $\frac{1}{3}$, or .33333.. or $\overline{3}$.	The word "rational" is a derivation of "ratio", and rational numbers are numbers that can be written as a ratio of two integers. "Q" stands for quotient.
Irrationals	Numbers that cannot be expressed as a fraction, such as π, $\sqrt{2}$, e. (We'll learn about these later).	If something is "irrational", it's not easy to explain or understand.
Real Numbers ($\mathbb{R}$)	Rational numbers and Irrational Numbers. The real number system can be represented on a number line:  	If a number exists on a number line that you can see, it must be "real". Note that the "smallest" real number is negative (–) infinity (−∞), and the largest real number is infinity (∞). We can never really get to these "numbers" (−∞ and ∞), but we can indicate them as the "end" of the real numbers.
Complex Numbers ($\mathbb{C}$)	Real numbers, plus imaginary numbers (concept only, such as $\sqrt{-2}$).	"Imaginary" numbers are difficult to imagine, since they are so "complex".


ℕ
ℤ
ℚ
ℝ
ℂ

Advantages

Perform operations on a collection of elements in a very **efficient** and **organized** manner

Conclusions

Sets are basic, fundamental data structures, with:

- unique elements
- no duplicates
- unchanging
- fixed or infinite number of elements

Im confused. Does it mean a set is similar to a Python set? Or what is the difference?

Sets vs. Python Set

In computer science (CS), a set is an abstract data type that can store unique values, without any particular order. It is a computer implementation of the mathematical concept of a finite set.

Mathematical vs. Python Sets

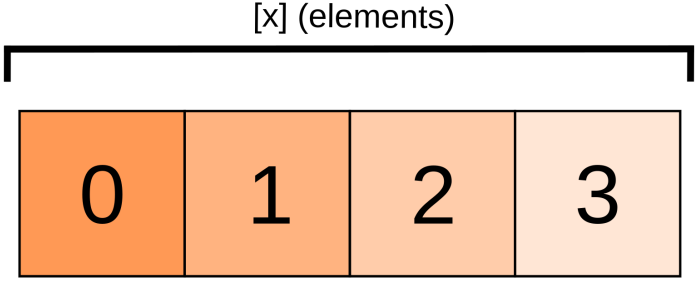
- Mathematical finite set: $\{1,2,3,4\}$
- Python set: $S = \{1,2,3,4\}$

Arrays

Arrays

In computer science, an **array** is a data structure consisting of a collection of elements, each identified by an **index** or a **key**. The simplest type of such data structure is a linear array, the one-dimensional array.

Typical "1 Dimensional" array



Element indexes are typically defined in the format *[x]*
[x] being the number of elements
For example: this array could be defined as *array[4]*

Arrays

Arrays are among the oldest and most important data structures, and are used by almost every program and programming language. They are also used to implement many other data structures, such as lists.

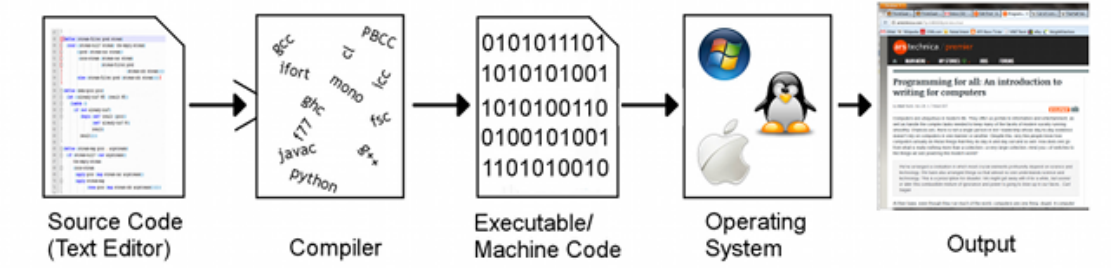
Arrays

Arrays are useful because the element indices can be computed at run time. Among other things, this feature allows a single iterative statement to process arbitrarily many elements of an array. For that reason, the elements of an array data structure are required to have the same size and should use the same data representation.

Run-time?

Runtime, run time, or execution time is the final phase of a computer program's life cycle, in which the code is being executed on the computer's central processing unit (CPU) as machine code. In other words, "runtime" is the running phase of a program.

Remember this? From source code to executable



Basic Array Operations

- traversal of an array
- access element X in an array
- searching element X in an array
- sorting an array

Example 1: Traversing the array A

```
1: procedure GETARRAY(A)                ▷ Returns the max value in A
2:     L ← length(A)
3:     for i=0 to L-1 do
4:         print A[i]
5:     end for
6: end procedure
```

Traversing the array

A = [1, 2, 3, 4, 5, 6, 7,8,9]

```
# Traversing the array
for element in A:
    print(element, end="-")}
```

Traversing the array, version 2

```
A = [1, 2, 3, 4, 5, 6, 7,8,9]

# Traversing the array
for i in range(len(A)):
    print(A[i], end="-")
```

Example 2: Find the max value in the array A

```
1:  procedure MaxArray(A)                    ▷ Returns the max value in A
2:      N ← length(A)
3:      MAX ← A[0]
4:      for from i=1 to N-1 do
5:          if A[i] > MAX then
6:              MAX = A[i]                   ▷ The MAX is A[i]
7:          end if
8:      end for
9:      return MAX
10: end procedure
```

Example 3: Search element X in array A

```
1:  procedure SEARCHARRAY(A)   ▷ Returns the max value in A
2:      X ← MyElement
3:      N ← length(A)
4:      for from i=0 to N-1 do
5:          if X = A[i] then
6:              return i                    ▷ The index for my match
7:          end if
8:      end for
9:      return -1                           ▷ otherwise return -1
10: end procedure
```

Search element in array

A = [1, 2, 3, 4, 5, 6, 7,8,9]

```
def find_element(A, n, key):
    for i in range(n):
        if A[i] == key:
            return i
    return -1
```

There are numerous applications of arrays

- Storing data in databases. Storing a list of customer names.
- Traffic Management.Traffic management systems use arrays to track vehicles and their flow. By analyzing data stored in arrays, traffic control centers can implement efficient signal timings and manage congestion effectively.

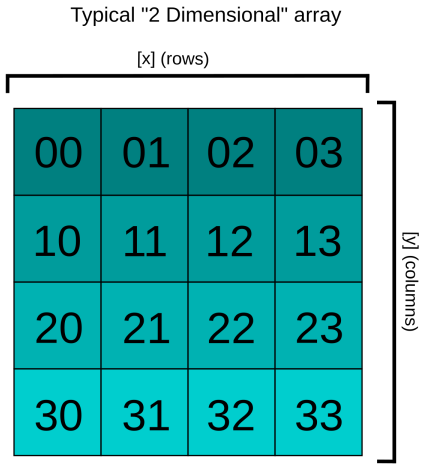
- Financial Analysis. It keeps track of various financial instruments, including stocks, bonds, and mutual funds. By organizing data in arrays, companies can perform analyses and make predictions easier
- Machine Learning. Machine learning algorithms often accept arrays as input, helping to train models and make predictions.

Matrices

What is a matrix?

In mathematics, a matrix (pl.: matrices) is a rectangular array or table of numbers, symbols, or expressions, with elements or entries arranged in rows and columns, which is used to represent a mathematical object or property of such an object.

Matrices



Element indexes are typically defined in the format*[x][y]*
[x] being the number of rows
[y] being the number of columns

Matrices

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ \vdots \\ m \end{array} \begin{array}{c} \begin{array}{cccc} 1 & 2 & \cdots & n \end{array} \\ \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ a_{31} & a_{32} & \dots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{m1} & a_{m2} & \dots & a_{mn} \end{bmatrix} \end{array}$$

Matrix multiplication

$$\begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \times \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix}$$

Matrix multiplication M x N

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \times \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1p} \\ b_{21} & b_{22} & \cdots & b_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ b_{n1} & b_{n2} & \cdots & b_{np} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & \cdots & c_{1p} \\ c_{21} & c_{22} & \cdots & c_{2p} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \cdots & c_{mp} \end{bmatrix}$$

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \cdots + a_{in} + b_{nj} = \sum_{k=1}^{n} a_{ik}b_{kj}$$

Basic Matrix Operations

- access X element in a matrix
- traversal of a matrix
- searching a matrix
- sorting a matrix

Accessing the elements of a matrix

```
A = [[1, 2, 3], [4, 5, 6], [7,8,9]]

# Accessing certain elements in a matrix
print("1st-element-of-1st-row:", A[0][0])
print("2nd-element-of-the-2nd-row:", A[1][2])
print("2nd-element-of-3rd-row:", A[2][1])
```

Traversing the matrix

```
A = [[1, 2, 3], [4, 5, 6], [7,8,9]]

# Traversing the matrix
for row in A:
    # Traversing the matrix
    for x in row:
        print(x, end="-")
    print()
```

There are numerous applications of matrices

- Encryption: Matrices encrypt data into unreadable formats and decode it for secure communication.
- Computer Graphics: transformations like scaling, rotation, and translation of objects in 2D and 3D graphics
- Machine Learning: fundamental data structures for neural networks

- Economics and Business: optimize business operations like supply chains and financial forecasting
- Navigation Systems: GPS systems use matrices to calculate positions, distances, and directions in 2D and 3D space.
- Weather Prediction: Matrices solve systems of differential equations to model and predict climate and weather patterns

Stacks

What is a stack?

The stack is an analogy to a set of physical items stacked one atop another, such as a stack of plates.

Lesson 3

Stacks

Stacks

Stacks are collections of elements, which support two main operations:

- **PUSH**: which adds an element to the collection
- **POP**: which removes the most recent elements from the collection

Stacks

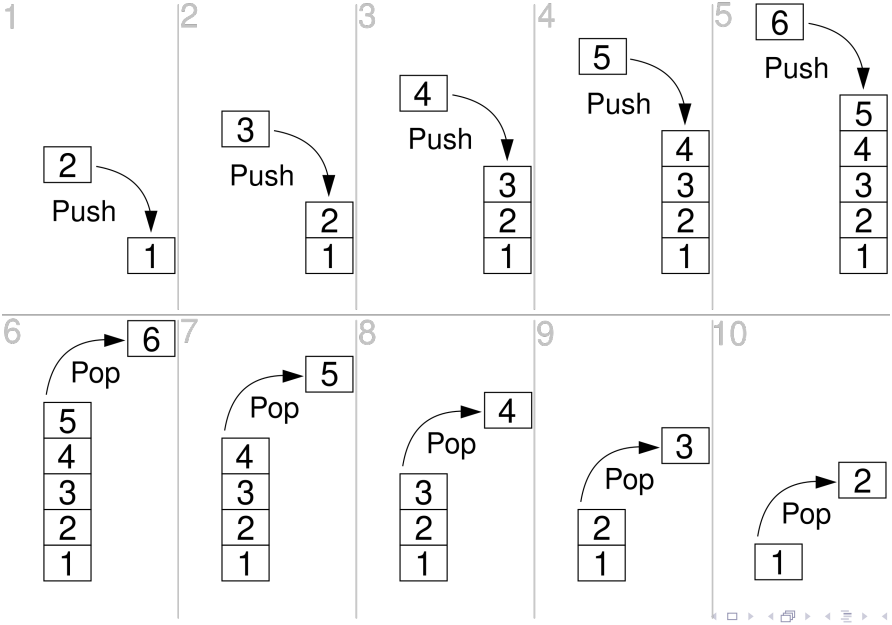
There might be, another operation called **PEEK**, which without modifying the stack, return the value of the last element added.

Stacks

- **PUSH**: which adds an element to the collection
- **POP**: which removes the most recent elements from the collection
- **PEEK**: returns the value of the last element added

Stacks

The stack supports few operations. And it operates in a certain, predefined order. The order in which an element added to or removed from a stack is described as last in, first out, referred to by the acronym **LIFO**


1
2
Push
1
2
3
Push
2
1
3
4
Push
3
2
1
4
5
Push
4
3
2
1
5
6
Push
5
4
3
2
1
6
6
Pop
5
4
3
2
1
7
5
Pop
4
3
2
1
8
4
Pop
3
2
1
9
3
Pop
2
1
10
2
Pop
1

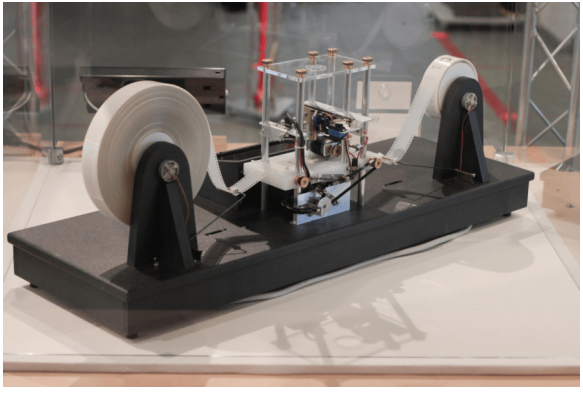
Stacks

"Stacks entered the computer science literature in 1946, when Alan Turing used the terms "bury" and "unbury" as a means of calling and returning from subroutines."

Who was Alan Turing?

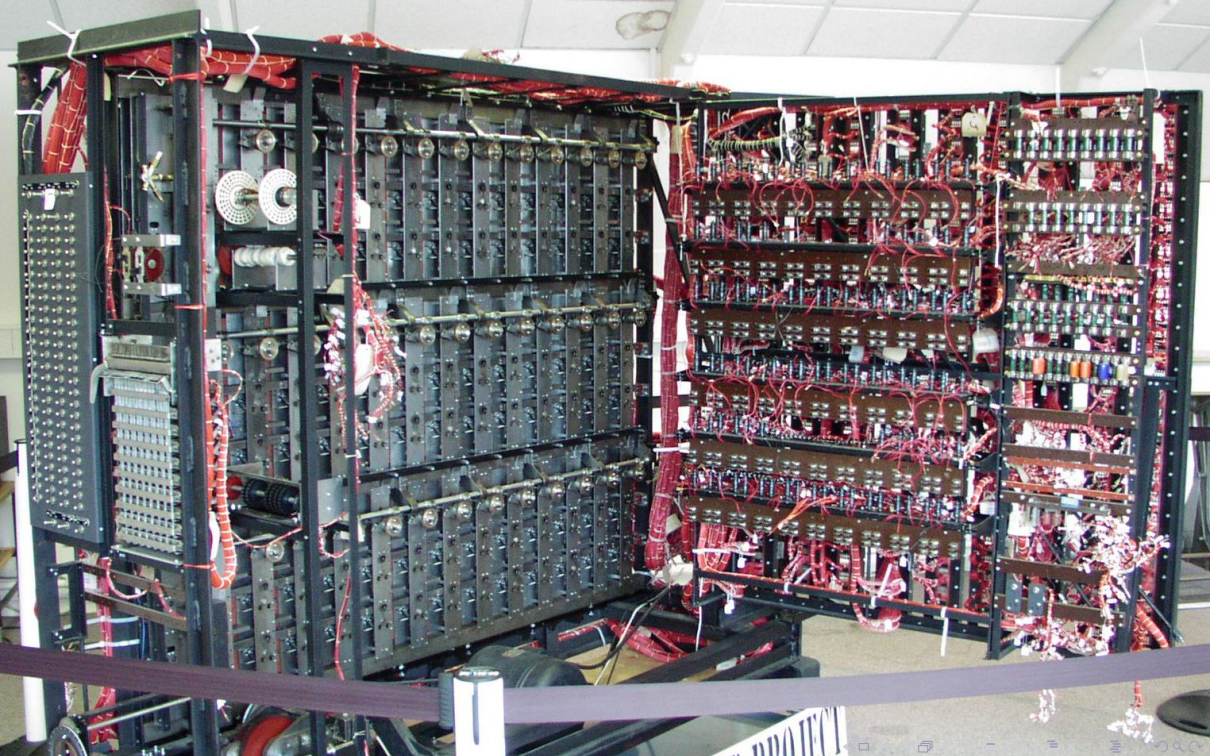
- The father of Computer Science
- English mathematician, computer scientist, logician, cryptanalyst
- Established the very first formalisation of the concepts of algorithm and computation with the Turing machine (a model of a general-purpose computer)

Remember this? The Turing Machine



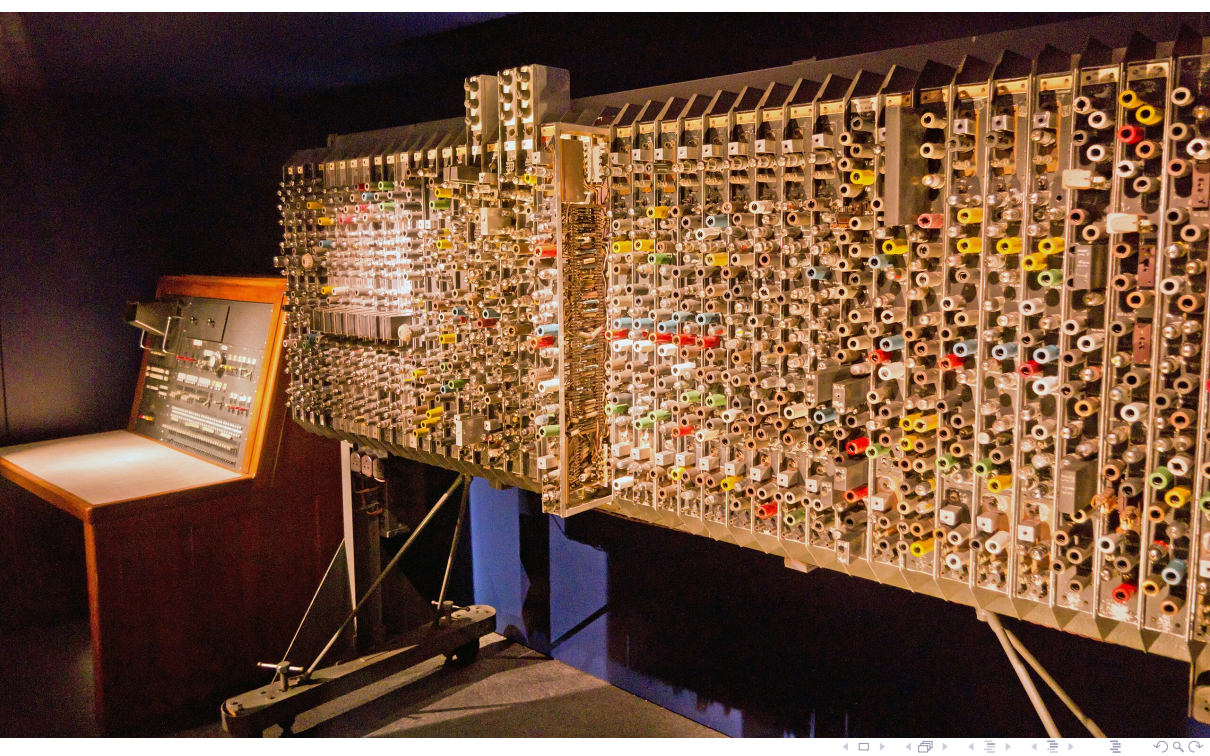
- Infinite tape, divided into cells with symbols
- A head can read/write symbols on the tape
- A register that stores the state of the machine

During World War II, Turing worked for the Government Code and Cypher School at Bletchley Park, Britain's codebreaking centre that produced Ultra intelligence. He led Hut 8, the section responsible for German naval cryptanalysis. Turing devised techniques for speeding the breaking of German ciphers, including improvements to the pre-war Polish bomba method, an electromechanical machine that could find settings for the Enigma machine.



Alan Turing - the inventor of ACE

Automatic Computing Engine (ACE) was a British early electronic serial stored-program computer



Want to know more? Read about Turing. See the **Imitation Game** movie.


THE
IMITATION GAME

Stacks

How can we implement a stack?

Stacks

A stack can be easily implemented using an array. The first element, usually at the zero offset, is the bottom, resulting in array[0] being the first element pushed onto the stack and the last element popped off.

Stacks

The program must keep track of the size (length) of the stack, using a variable top that records the number of items pushed so far, therefore pointing to the place in the array where the next element is to be inserted (assuming a zero-based index convention).

Stacks

The program must keep track of the size (length) of the stack, using a variable top that records the number of items pushed so far, therefore pointing to the place in the array where the next element is to be inserted (assuming a zero-based index convention).

Stack using arrays

```
# Create a stack. It initializes size of stack as 0
def createStack():
    stack = []
    return stack
```

```
# Stack is empty when stack size is 0
def isEmpty(stack):
    return len(stack) == 0

# Add an item to stack. It increases size by 1
def push(stack, item):
    stack.append(item)
    print(item + "-pushed-to-stack-")
```

```
# Remove an item from stack. It decreases size by 1
def pop(stack):
    if (isEmpty(stack)):
        # return minus infinite
        return str(-maxsize -1)
    return stack.pop()

# Return the top from stack without removing it
def peek(stack):
    if (isEmpty(stack)):
        # return minus infinite
        return str(-maxsize -1)
    return stack[len(stack) - 1]
```

Stack using LiFoQueue

```
from queue import LifoQueue
stack = LifoQueue()
stack.put(1)
stack.put(2)
stack.put(3)
print(stack.get()) # Prints 3
print(stack.get()) # Prints 2
```

There are numerous applications of stacks

- Undo mechanism in text editors
- Fwd and back buttons on web browsers
- Memory management in computer programming (static memory allocation. It can be used to keep track of functions calls)
- Implementing recursion

But what is recursion?

Recursion

- It is a programming technique, a way to program and develop a program where a function calls itself many times
- You take a big problem, into smaller problems, trying to apply a solution to solve the small problems
- You define a problem in terms of itself

Recursion

You are standing in a long queue of people. You must answer, how many people are behind you in the line?

Note: One person can see only the person standing directly in front and behind. One cannot look back and count. Each person is allowed to ask questions from the person standing in front or behind.

Recursion

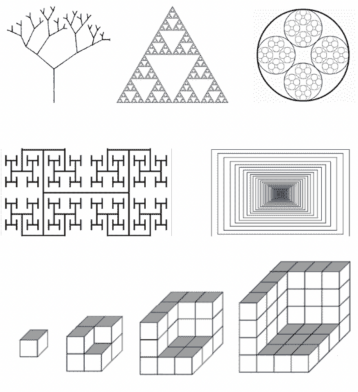
You look behind and see if there is a person there. If not, then you can return the answer "0". If there is a person, repeat this step and wait for a response from the person standing behind. Once a person receives a response, they add 1 and respond to the person that asked them or the person standing in front of them.

Recursion example

```
1: procedure personCount(currPerson)
2:     if noOneBehind(currPerson) == TRUE then
3:         return 0
4:     else personBehind == currPerson.checkBehind
5:         return 1 + personCount()
6:     end if
7: end procedure
```

Thinking recursively

- Learning to look for big things
- That are made from smaller things



Solving a problem recursively

Recursion is a function calling itself until a generic, base condition is true to produce the correct output. In other words, to solve a problem, we solve a problem that is a smaller instance of the same problem, and then use the solution to that smaller instance to solve the original problem.

Factorial number

In mathematics, the factorial of a non-negative integer $n!$ is the product of all positive integers less than or equal to n

$$n! = n \times (n-1)! \tag{1}$$

$$n! = n \times (n-1) \times (n-2) \times \ldots 3 \times 2 \times 1 \tag{2}$$

$$5! = 5 \times 4! = 5 \times 4 \times 3 \times 2 \times 1 = 120.$$

n	$n!$
0	1
1	1
2	2
3	6
4	24
5	120
6	720
7	5 040
8	40 320
9	362 880
10	3 628 800
11	39 916 800
12	479 001 600
13	6 227 020 800
14	87 178 291 200
15	1 307 674 368 000
16	20 922 789 888 000
17	355 687 428 096 000
18	6 402 373 705 728 000
19	121 645 100 408 832 000
20	2 432 902 008 176 640 000
25	$1.551\,121\,004 \times 10^{25}$
50	$3.041\,409\,320 \times 10^{64}$

Recursion example

```
1: procedure FACTORIAL(N)
2:     if N == 0 then
3:         return 1
4:     else
5:         return N * FACTORIAL(N − 1)
6:     end if
7: end procedure
```

Factorial, using recursion

```
def factorial (n):
    if n == 1:
        return 1
    else:
        return n * factorial (n-1)

print( factorial (5))
```

Solving a problem recursively

For a recursive algorithm to work, smaller subproblems must be found and arrive at the base case. In simple words, any recursive algorithm has two parts: the base case and the recursive structure.

The Base Case

The base case is a terminating condition where a function immediately returns the result. This is the smallest version of the problem for which we already know the solution.

The Recursive structure

The recursive structure is an idea to design a solution to a problem via the solution of its smaller sub-problems, i.e., the same problem but for a smaller input size. We continue calling the same problem for smaller input sizes until we reach the base case of recursion.

How recursion works?

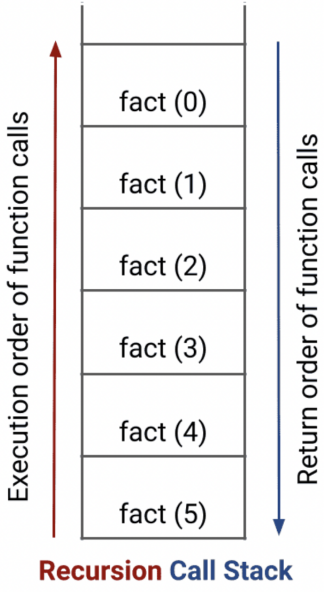
If we draw the flow of recursion for the factorial program, one can find this pattern: we are calling fact(0) last, but it is returning the value first. Similarly, we are calling fact(n) first, but it is returning the value last. Its a Last In First Out (LIFO) order for all recursive calls and return values.

Recursion uses Stacks behind

Order of recursive calls: larger problem to smaller problem
$fact(n) \rightarrow fact(n-1) \rightarrow ... \rightarrow fact(1) \rightarrow fact(0)$

Order of return values: smaller problem to larger problem
$fact(0) \rightarrow fact(1) \rightarrow ... \rightarrow fact(n-1) \rightarrow fact(n)$

Execution call stack!



Recursion is important!

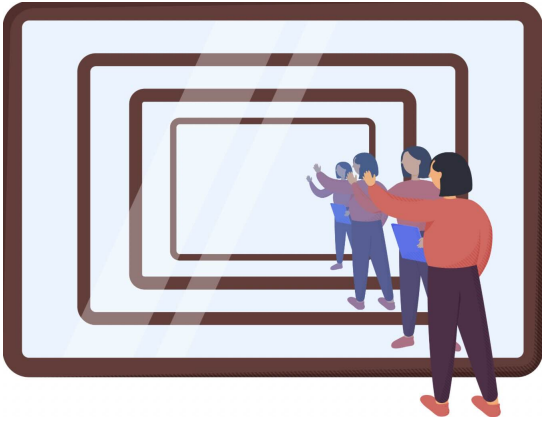
- the basis for Dynamic Programming and Divide and Conquer algorithms
- helps in solving complex problems by breaking them into smaller subproblems
- fundamental to sorting, like quicksort, mergesort
- used in traversing trees and other complex data structures

Recursion vs Iteration?

A program is called **recursive** when an entity calls itself. A program is called **iterative** when there is a loop (or repetition) of some sort.

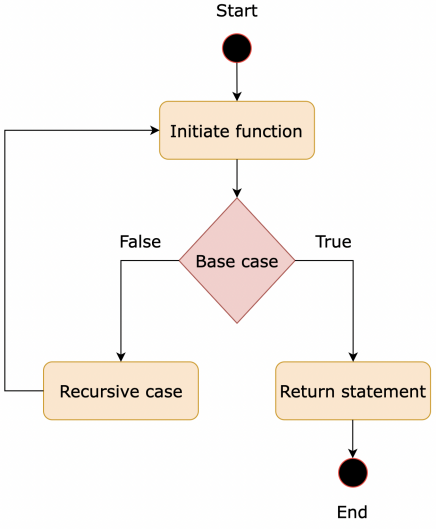
Recursion

- each function call creates a smaller problem to solve
- and these calls continue until reaching a basic case which is trivial to solve



Recursive function

- **Base case** The condition under which the function stops calling itself. This prevents infinite recursion and provides a direct answer for the simplest instance of the problem.
- **Recursive case** The part of the function that reduces the problem into smaller instances and calls itself with small data.



Recursive Factorial

```
def factorial (n):
    # Base case: if n is 1 or 0, factorial is 1
    if n == 0 or n == 1:
        return 1
    # Recursive case: n * factorial of (n-1)
    else:
        return n * factorial (n - 1)
print( factorial (5))
```

Recursion Types

- direct recursion
- indirect recursion
- tail recursion

Direct Recursion - the simplest form of recursion. A function directly calls itself within its definition.

```
def direct_recursion (n):
    if n <= 0:
        return
    print(n)
    direct_recursion (n - 1)
```

Indirect Recursion - creates a cylces of function calls. A function calls another function, which calls the original function.

```
def functionA(n):
    if n <= 0:
        return
    print(n)
    functionB(n - 1)

def functionB(n):
    if n <= 0:
        return
    print(n)
    functionA(n - 2)
```

Tail Recursion - the recursive call is the last operation in the function.

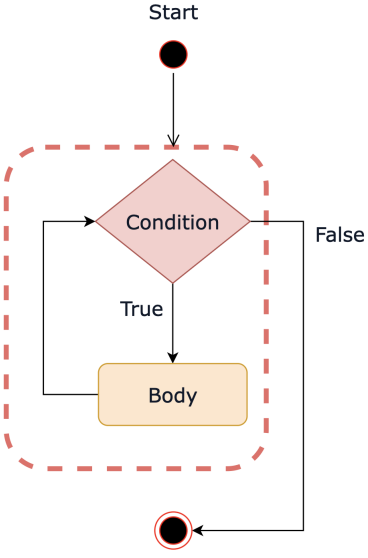
```
def tail_recursion (n, aggregate=1):
    if n == 0:
        return aggregate
    return tail_recursion (n − 1, n ∗ aggregate)
```

Iterative

- a repetitive process
- runs a number of times until a specified condition is met
- or certain number of iterations have been reached

Iterative function

- **Initialization** Start with initializing variables or setting initial conditions required for the iterative process
- **Condition** Check a condition that determines whether the iteration should continue or stop
- **Body** Execute a set of instructions or operations that represent the core logic of the iteration.



How to build an iterative function?

- loops: while, do-while, for loops
- for loop: when the number of iterations is known beforehand
- while loop: when the number of iterations is unknown, and the loop continues as long as a condition is true
- do-while loop: same as while loop but ensures that the loop is executed at least once before the condition is tested

Iterative Factorial - the iterative function to calculate the factorial

```
def factorial (n):
    result = 1
    for i in range(1, n + 1):
        result *= i
    return result
print( factorial (5))
```

When to use recursion vs. iteration

The nature of the problem

- naturally fit a recursive structure or require breaking down into smaller subproblems
- for loop: when the number of iterations is known beforehand
- use iteration for problems with a straightforward repetitive process or when the number of iterations is known beforehand

When to use recursion vs. iteration

Complexity and readability

- Use recursion when it simplifies the code and makes it more readable
- Use iteration when recursion would make the code unnecessarily complex or when avoiding the risk of stack overflow is essential

When to use recursion vs. iteration

Performance consideration

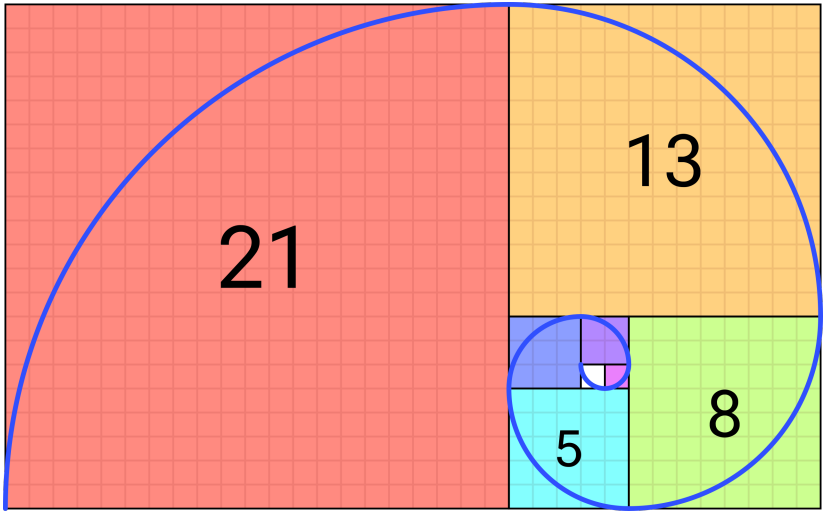
- Use recursion if the problem's recursive nature allows for more elegant and maintainable code and manageable stack usage
- Use iteration if memory usage is a concern and the problem can be solved efficiently with loops.

Use recursion

- **Divide and conquer** problems like merge sort, quicksort, and binary search where the problem is divided into smaller subproblems that are solved recursively
- **Tree and graph problems** Traversals (e.g., in-order, pre-order, post-order for trees) and pathfinding in graphs that naturally fit a recursive approach.
- **Dynamic programming** Problems like the Fibonacci sequence, where recursion with memoization simplifies the solution
- **Combinatorial problems** Generating permutations and combinations and solving puzzles like the Tower of Hanoi

Use iteration

- **Simple loops** Problems like summing a list of numbers, iterating through arrays or lists, and simple counting problems
- **Linear problems** Iterative solutions for problems requiring a linear scan, such as finding an array's maximum or minimum value
- **Repetitive tasks** Problems that require repeating a task a known number of times, such as printing numbers from 1 to N or iterating through data structures like arrays and linked lists.


21
13
8
5

Fibonacci sequence

Fibonacci sequence

In mathematics, the **Fibonacci sequence** is a sequence in which each element is the sum of the two elements that precede it. Numbers that are part of the Fibonacci sequence are known as **Fibonacci numbers**, commonly denoted F_n. Many writers begin the sequence with 0 and 1, although some authors start it from 1 and 1[1][2] and some (as did Fibonacci) from 1 and 2. Starting from 0 and 1, the sequence begins

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ... (sequence A000045 in the OEIS)

Fibonacci sequence

Definition [edit]

The Fibonacci numbers may be defined by the recurrence relation[7]

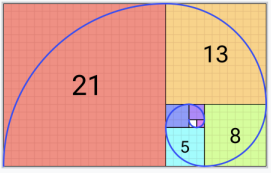
$$F_0 = 0, \quad F_1 = 1,$$

and

$$F_n = F_{n-1} + F_{n-2}$$

for $n > 1$.

Under some older definitions, the value $F_0 = 0$ is omitted, so that the sequence starts with $F_1 = F_2 = 1$, and the recurrence $F_n = F_{n-1} + F_{n-2}$ is valid for $n > 2$.[8][9]



The Fibonacci spiral: an approximation of the golden spiral created by drawing circular arcs connecting the opposite corners of squares in the Fibonacci tiling (see preceding image)

The first 20 Fibonacci numbers F_n are:

F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}	F_{16}	F_{17}	F_{18}	F_{19}
0	1	1	2	3	5	8	13	21	34	55	89	144	233	377	610	987	1597	2584	4181

Fibonacci Recursion

```
def fibonacci_recursive (n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci_recursive (n-1) + fibonacci_recursive (n-2)

# Main Code
n = 10
print(f" Recursive method: Fibonacci({n}) = { fibonacci_recursive (n)}
```

Fibonacci Iterative

```
def fibonacci_iterative (n):
    if n <= 0:
        return 0
    elif n == 1:
        return 1
    a, b = 0, 1
    for _ in range(2, n + 1):
        a, b = b, a + b
    return b

# Main code
n = 10
print(f" Iterative method:Fibonacci({n}) ={ fibonacci_iterative (n)}
```

Advantages of Stacks

- Simplicity: Stacks are a simple and easy-to-understand data structure
- Efficiency: Push and pop operations on a stack are very fast, providing efficient access to data.
- LIFO: Stacks follow the LIFO principle, ensuring that the last element added to the stack is the first one removed.
- Limited memory usage: Stacks only need to store the elements that have been pushed onto them, making them memory-efficient compared to other data structures.

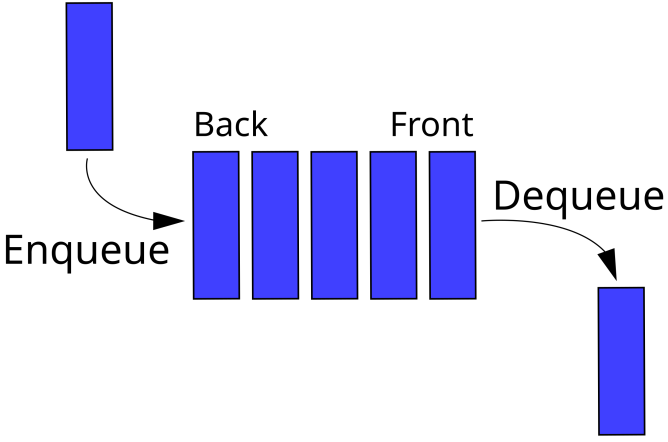
Disadvantages of Stacks

- Limited access: Elements can only be accessed from the top, making it difficult to retrieve or modify elements in the middle of the stack.
- Overflow: If more elements are pushed onto a stack than it can hold, an overflow error will occur, resulting in a loss of data.
- No random access: Stacks do not allow for random access to elements, making them unsuitable for applications where elements need to be accessed in a specific order.
- Limited capacity: Fixed capacity, a limitation if the number of elements that need to be stored is unknown or highly variable.

Queues

Queues

A queue is a collection of entities that are maintained in a sequence and can be modified by the addition of entities at one end of the sequence and the removal of entities from the other end of the sequence.


Back
Front
Dequeue
Enqueue

By convention, the end of the sequence at which elements are added is called **the back, tail, or rear** of the queue, and the end at which elements are removed is called the **head or front** of the queue, analogously to the words used when people line up to wait for goods or services.

Queues

The operations of a queue make it a first-in-first-out **FIFO** data structure. In a FIFO data structure, the first element added to the queue will be the first one to be removed.
A queue is an example of a linear data structure, or more abstractly a sequential collection.

Queue Properties

- **Front:** Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the front of the queue (head of the queue)
- **Rear:** Position of the last entry in the queue, that is, the one most recently added, is called the rear of the queue. (tail of the queue)
- **Size:** the current number of elements in the queue
- **Capacity:** the maximum number of elements the queue can hold

Queue Operations

- **Enqueue** — Add an element to the end of the queue
- **Dequeue** — Remove an element from the front of the queue
- **Is Empty** — Check if the queue is empty
- **Is Full** — Check if the queue is full
- **Peek** — Get the value of the front element without removing it

Queue implements a basic FIFO queue

```
from queue import Queue
q = Queue()
q.put(1) # Add 1 to queue
q.put(2)
q.put(3)
print(q.qsize()) # Prints 3
print(q.get()) # Prints 1
print(q.get()) # Prints 2
```

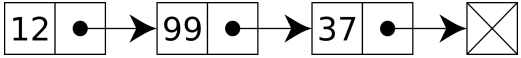
Python Queue

```
from queue import Queue
q = Queue()
# The key methods available are:
# qsize() – Get the size of the queue
# empty() – Check if queue is empty
# full() – Check if queue is full
# put(item) – Put an item into the queue
# get() – Remove and return an item from the queue
# join() – Block until all tasks are processed
```

Linked lists

Linked Lists

A linked list is a linear collection of data elements where each element points to the next. It is a data structure consisting of a collection of nodes which together represent a sequence.


12
99
37

Linked Lists

each node contains data, and a reference (in other words, a link) to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence during iteration.

Linked Lists

A drawback of linked lists is that data access time is linear in respect to the number of nodes in the list. Because nodes are serially linked, accessing any node requires that the prior node be accessed beforehand

Linked Lists

Linked lists are among the simplest and most common data structures. They can be used to implement several other common abstract data types, including stacks, queues, associative and dynamic arrays.

Associative Arrays

Dynamic Arrays

Variable-size list data structure that allows elements to be added or removed. Dynamic arrays overcome a limit of static arrays, which have a fixed capacity that needs to be specified at allocation.

Associative Arrays

A map, symbol table, or dictionary is an abstract data type that stores a collection of (key, value) pairs, such that each possible key appears at most once in the collection. It supports lookup, remove, and insert operations.

Linked Lists

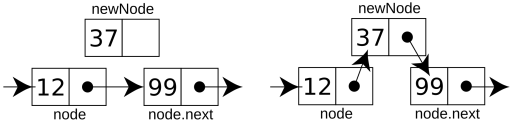
The principal benefit of a linked list over a conventional array is that the list elements can be easily inserted or removed without reallocation or reorganization of the entire structure because the data items do not need to be stored contiguously in memory or on disk, while restructuring an array at run-time is a much more expensive operation.

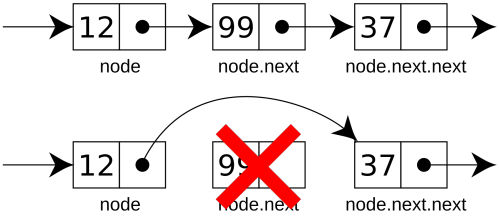
Linked Lists

Linked lists allow insertion and removal of nodes at any point in the list, and allow doing so with a constant number of operations by keeping the link previous to the link being added or removed in memory during list traversal.

Linked list Operations

- **INSERT** — Add an element to the end of the queue
- **DELETE** — Remove an element from the front of the queue
- **TRAVERSAL** — Check if the queue is empty

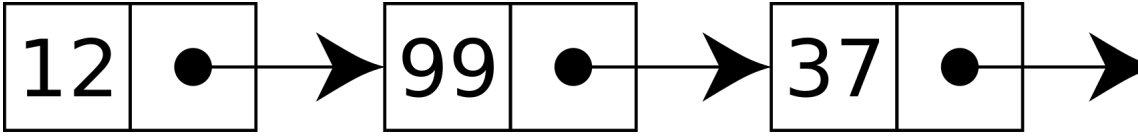

newNode
37
12
99
node
node.next
newNode
37
12
99
node
node.next


12
99
37
node
node.next
node.next.next
12
37
node
node.next.next

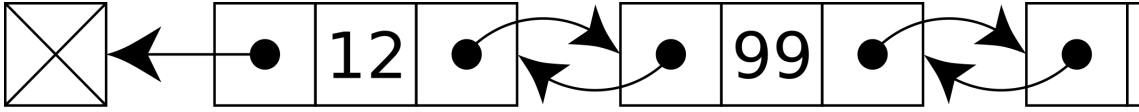
Linked lists

- **Single linked list** lists contain nodes which have a value and a next pointer
- **Double linked list** contains nodes which inlcudes the next and previous pointer links
- **circular linked list** a list where the last nide has a pointer to the first node

Single Linked list



Double Linked list



Circular Linked list