QUESTION 1

Computer advancements essay.

Charls Babbage, an English polymath of the 19th century, laid the groundwork for modern computing with his designs for the Difference Engine and the Analytical Engine. Although these mechanical marvels were never fully realized during his lifetime, they introduced the concept of a programmable device, planting the seeds for future generations of computer engineers and mathematicians.

Fast-forward to the early 20th century, and we find pioneers like Alan Turing at Bletchley Park in England, where the British governement established a secret facility for codebreaking during WWII. Turing's theoretical work, particularly on the concept of a universal machine, fueled the development of electronic computers. Around the same time in the United States, John von Neumann was a vital figure at the Institute for Advanced Study in Princeton, formulating the "von Neumann architecture" that still underpins the structure of modern computers, with memory storing both data and program instructions.

The hardware side witnessed its first major leap with vacuum tube-based machines like the ENIAC "Electronic Numerical Integrator and Computer", built at the University of Pennsylvania in 1945. Though enormous and prone to overheating, ENIAC could perform calculations far faster than any purely mechanical counterpart. The invention of the transistor at Bell Labs in 1947 by John Bardeen, Walter Brattain, and William Shockley launched a revolution in miniaturization, leading to machines like the IBM 1401 in the late 1950s. By the 1970s, integrated circuits had shrunk hardware further, enabling personal computers such as the MITS Altair 8800 to appear in homes.

In parallel, software evolved from machine code to more intuitive programming languages. Grace Hopper, working at Harvard University in the 1940s, championed the development of compilers, which translated humna-readable code into instructions for the machine. This paved the way for languages like FORTRAN in 1957, created by John Backus and his team at IBM, and COBOL in 1959, influenced heavily by Hopper's ideas. The personal computing boom of the 1980s and 1990s, propelled by figures like Bill Gates "Microsoft" and Steve Jobs "Apple", popularized high-level languages such as C, C++, and eventually Java, expanding the software ecosystem into new realms.

Today, computing hardware has advanced to the nano-scale, packing billions of transistors onto a single chip, while software encompasses everything from operating systems like Linux and Windows to AI-driven technologies. Though the digital world has evolved beyond anything Babbage could have imagined, the roots of modern computing trace right back to that brilliant investor's vision of a machine that could perform calculations automatically.

USED REMARKABLE 2 FOR THIS ASSESSMENT.

(a) 1 0 1 1 0 0 1

$2^6\ 2^5\ 2^4\ 2^3\ 2^2\ 2^1\ 2^0$

[64 0 16 8 0 0 1]

ADDING ALL TOGETHER

89

(b) 1 0 1 1 . 0 1 0 1

$2^3\ 2^2\ 2^1\ 2^0\ \ 2^{-1}\ 2^{-2}\ 2^{-3}\ 2^{-4}$

[8 0 2 1] . [0 1/4 0 1/16]

ADDING ALL

11.3125

(c)

246 : 2 = 123 → 0

123 : 2 = 61 → 1

61 : 2 = 30 → 1

30 : 2 = 15 → 0

15 : 2 = 7 → 1

7 : 2 = 3 → 1

3 : 2 = 1 → 1

1 : 2 = 0 → 1

11110110

BOTTOM TO TOP TO GET THE RESULT.

(d) 41B2

RECALL THAT IN HEXADECIMAL, B REPRESENTS 11 IN DECIMAL.

$4 \times 16^3 = 16384$

$1 \times 16^2 = 256$

$11 \times 16^1 = 176$

$2 \times 16^0 = 2$

SUM ALL UP TO GET

16818

QUESTION 2

(a)

```
  1 1 1 1 1
  1 1 0 0 1 1  +
    1 1 1 1 1
-------------
  1 0 1 0 0 1 0
```

(b) SHIFT THE DECIMAL POINT SO THAT THERE IS EXACTLY ONE NON-ZERO DIGIT TO THE LEFT OF IT.

$0.00435 = 4.35 \times 10^{-3}$

(c)

- VALUE TO BINARY

```
+44 : 2 = 22 → 0
 22 : 2 = 11 → 0
 11 : 2 =  5 → 1
  5 : 2 =  2 → 1
  2 : 2 =  1 → 0
  1 : 2 =  0 → 1
```

+44 = 101100

- +44 IN 32-BIT

00000000 00000000 00000000 00101100

- INVERT ALL THE BITS TO GET ONE'S COMPLEMENT

11111111 11111111 11111111 11010011

- ADD 1 TO GET THE TWO'S COMPLEMENT

```
11111111 11111111 11111111 11010011  +
                                    1
-------------------------------------
11111111 11111111 11111111 11010100
```

↳ THIS RESULT IS THE 32-BIT TWO'S COMPLEMENT OF −44

(d)

- MINUEND = 1100011
- SUBTRAHEND = 0110111

- FIND THE TWO'S COMPLEMENT OF THE SUBTRAHEND

ONE'S COMPLEMENT → 1001000 +

THEN ADD 1 1 =

1001001

- ADD THE MINUEND TO THE TWO'S COMPLEMENT

$$\begin{array}{r} 1100011 \\ +\ 1001001 \\ \hline 0101100 \end{array}$$

(with carry 1 1)

* IN TWO'S COMPLEMENT ARITHMETIC THE LEFTMOST BIT IS DISCARDED - WHEN STAYING WITHIN THE SAME FIXED WIDTH.

* IN DECIMAL THAT CORRESPONDS TO +44.

QUESTION 3

(a)

- *London is the capital of France*:
  This is a declarative sentence making an assertion that can be assigned a truth value. However, the statement happens to be false, because Paris is the capital of France. Hence, this is a **proposition**, and its truth value is false.
- *Open the window please*:
  This is a request/command, not a declarative statement. it does not assert anything that can be true or false. Thus, **this is not a proposition** - having no truth value in the classical logical sense.
- *X + 5 = 10*:
  As stated, it depends all on the value of X. It is known as an open statement or predicate, rather than a proposition. It cannot be assigned a definite true of false without knowing X. Consequently, **this is not a proposition** in its current form.

(b)

p : "THE TABLE IS GREEN."
q : "THE BALL IS YELLOW."

- IF THE TABLE IS GREEN, THEN THE BALL IS YELLOW.

$$p \Rightarrow q$$

- THE TABLE IS NOT GREEN OR THE BALL IS YELLOW.

$$\neg p \vee q$$

- IT IS NOT TRUE THAT THE TABLE IS GREEN AND THE BALL IS YELLOW.

$$\neg(p \wedge q)$$

(c)

STATEMENTS

S1: Charlie is not a cook. $\neg C$

S2: Alice is an architect or Bob is a builder. $A \vee B$

S3: If Bob is a builder, then Charlie is a cook. $B \Rightarrow C$

i) ALICE IS AN ARCHITECT $S1 \wedge S2 \wedge S3 \vdash A$

FROM S1: $\neg C$ AND S3: $B \Rightarrow C$.

IF BOB WERE A BUILDER, THEN BY S3 CHARLIE IS A COOK.

BUT S1 SAYS $\neg C$. THEREFORE $B \Rightarrow C$ AND $\neg C$.

THINKING, CANNOT BOTH BE TRUE IF B WERE TRUE.

ASSUMING B, THEN FROM $B \Rightarrow C$, CONTRADICTING $\neg C$.

B MUST BE FALSE TO AVOID CONTRADICTIONS. $\neg B$

NOW RECALLING S2: $A \vee B$. BUT ESTABLISHING $\neg B$,

TO MAKE $A \vee B$ TRUE, WE MUST HAVE A. THUS,

FROM S1, S2, AND S3 TOGETHER, IT FOLLOWS THAT A.

IT IS LOGICALLY CORRECT.

ii) BOB IS A BUILDER $S1 \wedge S2 \wedge S3 \models B$

FROM S1: $\neg C$ AND S3: $B \Rightarrow C$

IF B WERE TRUE, THAT WOULD FORCE C, CONTRADICTING $\neg C$. SO $\neg B$ MUST HOLD INSTEAD. HENCE WE CANNOT CONCLUDE B. IN FACT, WE CONCLUDE THE OPPOSITE.

IT IS NOT LOGICALLY CORRECT

iii) CHARLIE IS A BUILDER

THERE IS NO STATEMENT IN THE PREMISES LINKING CHARLIE TO BE OR NOT BEING A BUILDER. THE STATEMENTS ONLY MENTION CHARLIE BEING A COOK OR NOT. THERE IS SIMPLY NO LOGICAL PATH FROM S1, S2, S3 TO DEDUCE WHETHER CHARLIE IS OR IS NOT A BUILDER.

IT IS NOT LOGICALLY CORRECT BECAUSE THERE IS NO BASIS TO CONCLUDE CHARLIE IS A BUILDER.

iv) CHARLIE IS NOT A BUILDER

FOR THE SAME REASONS AS iii), THE PREMISES GIVE US INFORMATION ONLY ABOUT:

- CHARLIE STATUS AS A COOK.
- BOB BEING A BUILDER OR NOT.
- ALICE BEING AN ARCHITECT OR NOT.

NO PREMISES SAYS "ONLY ONE OCCUPATION PER PERSON" OR "IF YOU ARE A COOK, YOU MUST BE A BUILDER". THERE IS NO RULE ABOUT CHARLIE'S POSSIBLE BUILDER STATUS.

IT IS NOT LOGICALLY CORRECT

QUESTION 4

(a)

- THE SET OF DATES IN THE MONTH OF MARCH.

$$\{d \in \mathbb{Z} : 1 \leq d \leq 31\}$$

THAT IS, THE SET OF ALL INTEGERS d SUCH THAT d IS BETWEEN 1 AND 31, INCLUSIVE.

- $\{17, 19, 21, 23, 25\}$.

NOTICE THESE ARE CONSECUTIVE ODD INTEGERS FROM 17 TO 25.

$$\{n \in \mathbb{Z} : n \text{ IS ODD AND } 17 \leq n \leq 25\}$$

- THE SET OF NON-NEGATIVE RATIONAL NUMBERS.

A RATIONAL NUMBER IS NON-NEGATIVE IF IT IS GREATER THAN OR EQUAL TO ϕ.

$$\{x \in \mathbb{Q} : x \geq \phi\}$$

b

Universal set and each subset:

U = (1, 2, 3, 4, 5, 6, 7, 12, blue, red, green, pink, water)
A = (blue, red, green, pink)
B = (12, 5, 6, 7)
C = (green, 4, 12, 7, water)
D = (1, 2, 5)

i) **(A U B) n C**

A U D = (blue, red, green, pink) U (1, 2, 5) = (blue, red, green, pink, 1, 2, 5)
(A U B) n C = (blue, red, green, pink, 1, 2, 5) n (green, 4, 12, 7, water)
The only common element of the intersection is green
Thus, **(A U B) n C = (green)**

---

ii) **(B n D) n A**

B n D = (12, 5, 6, 7) n (1, 2, 5) = (5)
(B n D) n A = (5) n (blue, red, green, pink) = 0
Thus, **(B n D) n A = 0**

---

iii) **D - B**

D - B = (1, 2, 5) - (12, 5, 6, 7) = (1, 2)
Thus, **D - B = (1, 2)**

---

iv) **A' n (B U D)'**

A' = U - A = (1, 2, 4, 5, 6, 7, 12, blue, red, green, pink, water) - (blue, red, green, pink) = (1, 2, 4, 5, 6, 7, 12, water)
B U D = (12, 5, 6, 7) U (1, 2, 5) = (1, 2, 5, 6, 7, 12)
Then (B U D)' = U - (B U D)' = (1, 2, 4, 5, 6, 7, 12, blue, red, green, pink, water) - (1, 2, 5, 6, 7, 12) = (4, blue, red, green, pink, water)
A' n (B U D)' = (1, 2, 4, 5, 6, 7, 12, water) n (4, blue, red, green, pink, water) = **(4, water)**

---

v) **A x D**

A x D = (blue, red, green, pink) x (1, 2, 5) = **((blue, 1), (blue, 2), (blue, 5), (red, 1), (red, 2), (red, 5), (green, 1), (green, 2), (green, 5), (pink, 1), (pink, 2), (pink, 5))**

(c)

The relationship ***R*** from ***P*** to ***Q*** is defined as the following:

(x, y) in **R** <--> "x is divisible by y with no remainder"

P = (21, 12, 5, 22)
Q = (4, 3)

-Checking 21 with elements in Q:

21 / 4 = 5 remainder 1. (not divisible)
21 / 3 = 7 remainder 0. (divisible)

-Checking 12 with elements in Q:

12 / 4 = 3 remainder 0. (divisible)
12 / 3 = 4 remainder 0. (divisible)

-Checking 5 with elements in Q:

5 / 4 = 1 remainder 1. (not divisible)
5 / 3 = 1 remainder 2. (not divisible)

-Checking 22 with elements in Q:

22 / 4 = 5 remainder 2. (not divisible)
22 / 3 = 7 remainder 1. (not divisible)

Comparing the pairs: ***R*** = ((21, 3), (12, 4), (12, 3))

Question 5

a

A = (c, a)
B = (a, b, c)

i) A x A = ((c, c), (c, a), (a, c), (a, a))

ii) B x A = ((a, c), (a, a), (b, c), (b, a), (c, c), (c, a))

iii) A x B = ((c, a), (c, b), (c, c), (a, a), (a, b), (a, c))

---

b

i) ((2, 2), (2, 3), (2, 4), (3, 2), (3, 3), (3, 4)) ... this set is not reflexive, not irreflexive, not symmetric, not antisymmetric, but **it is transitive.**

ii) ((1, 1), (1, 2), (2, 1), (2, 2), (3, 3), (4, 4)) ... this set **is reflexive**, not irreflexive, **is symmetric**, not antisymmetric, **and it is transitive.**

iii) ((1, 4), (4, 2)) ... this set is not reflexive, **is irreflexive**, not symmetric, **it is antisymmetric**, not transitive.

---

c

U = (1, 2, 3, 4, 5, 6, 7, 8, 9)
A = (1, 2, 3, 4, 5)
B = (4, 5, 6, 7)
C = (5, 6, 7, 8, 9)
D = (1, 3, 5, 7, 9)
E = (2, 4, 6, 8)
F = (1, 5, 9)

i) **A n (B U E):** (1, 2, 3, 4, 5) n (2, 4, 5, 6, 7, 8) = **(2, 4, 5)**

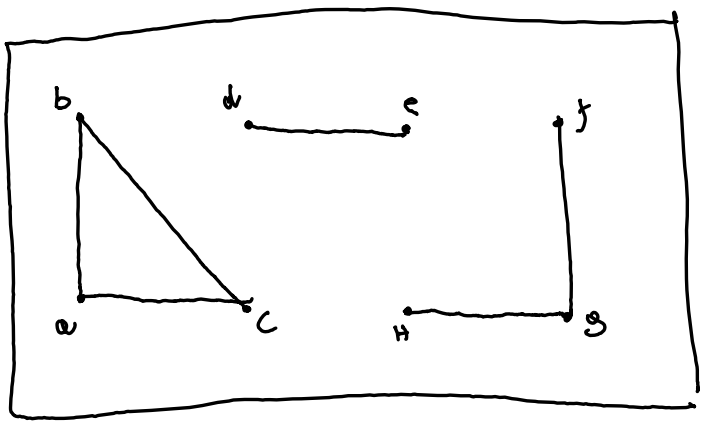
ii) **(B n F) U (C n E):** (5) U (6, 8) = **(5, 6, 8)**

iii) **B'** = **(1, 2, 3, 8, 9)**

iv) **(A n D) - B** = (1, 3, 5) - (4, 5, 6, 7) = **(1, 3)**

QUESTION 6

(a)



H

i) The degree of node a is 2.

ii) There are 8 vertices in the graph H.

iii) There are 3 connected components in H. The first (a, b, c), the second (d, e), and the third (f, g, h).

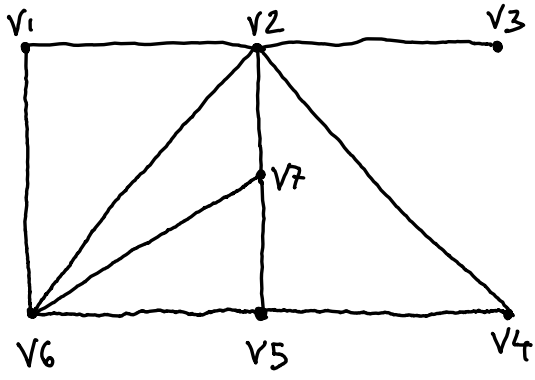
iv) It is the triangle on the left (a -> b -> c -> a).

v) A Hamiltonian cycle visits every vertices of H exactly once. The simple cycle visits only 3 of the 8 vertices, thus, it is not a Hamiltonian cycle.

(c) ADJACENCY MATRIX

$$A_H = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

(b)



G

i) Vertex 3 is the only pendant one.

ii) A path requires every pair of consecutive vertices be adjacent. While v1 is adjacent to v6, there's is no edge between v6 and v2. Thus (v1, v6, v2, v5, v4) sequence is not a path.

iii) A simple circle cannot repeat any vertices other than the first and last implying every pair of consecutive vertices must be adjacent. However, the sequence (v6, v7, v5, v6, v2, v1, v6) repeats v6 in the middle failing the condition.

iv) Yes, G is connected. It is possible to find a path between any two vertices on the graph.

v) v6 is adjacent to v5, v7 and v1.

(c) ADJACENCY MATRIX

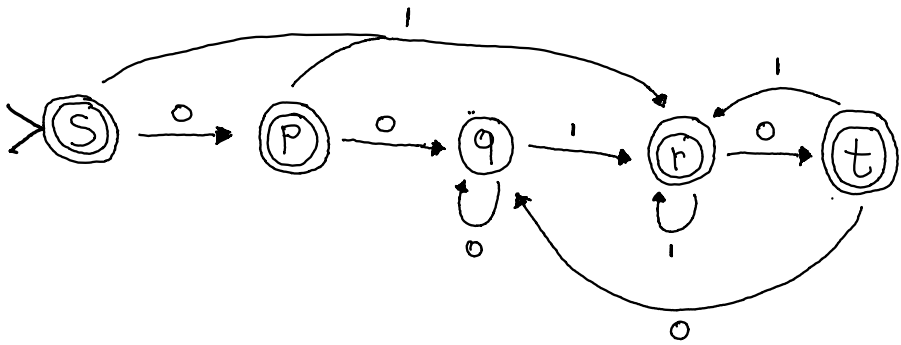
$$A_G = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

QUESTION 7

a)

	0	1
s	p	r
p	q	r
q	q	r
r	t	r
t	q	r

i)



ii)

-Checking **110100**:

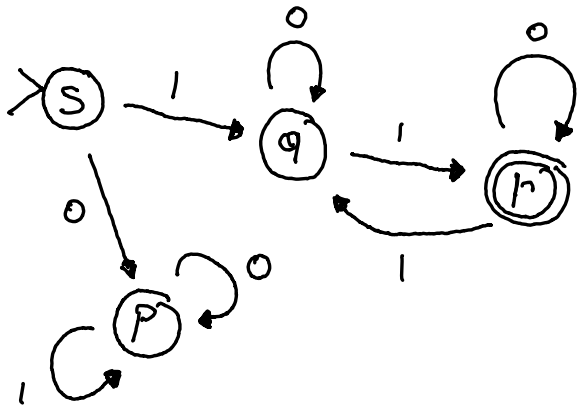
(s, 1) -> moves to r from s.
(r, 1) -> self-loop on r.
(r, 0) -> moves to t from r.
(t, 1) -> moves to r from t.
(r, 0) -> moves to t from r.
(t, 0) -> moves to q from t.
Since the string lands to q, **it is not accepted.**

-Checking **011010**:

(s, 0) -> moves to p from s.
(p, 1 -> moves to r from p.
(r, 1) -> self-loop on r.
(r, 0) -> moves to t from r.
(t, 1) -> moves to r from t.
(r, 0) -> moves to t from r.
Since the string lands to t, **it is accepted.**

iii) After running several tests, the automaton A accepts all the strings except those ones ending with more than one 0. Every string finishing with **00** or more, will land on the non-accepting state q.

b)



i)

	0	1
s	p	q
p	p	p
q	q	r
r	r	q

ii)

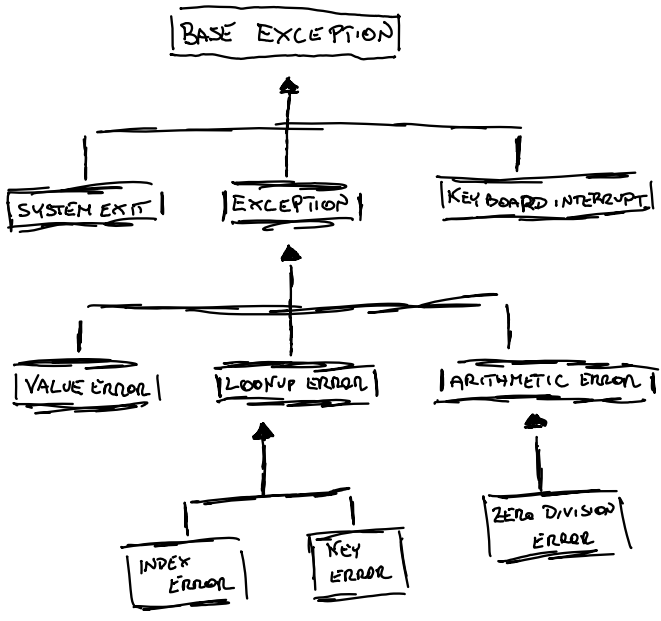
-Checking **01**:
(s, 0) -> moves to p from s.
(p, 1) -> self-loop on p.
Since the string falls into the trap state, **it is not accepted**.

-Checking **101**:
(s, 1) -> moves to q from s.
(q, 0) -> self-loop on q.
(q, 1) -> moves to r from q.
r is the accepting state, hence, the string **is accepted.**

-Checking **100**:
(s, 1) -> moves to q from s.
(q, 0) -> self-loop on q.
(q, 0) -> self-loop on q.
q is not an accepting state, the string **is not accepted**.

iii) At first glance, the automaton does not accept all the strings starting with **0,** because, each one of them will fall into the trap state making it impossible to leave once entered. Therefore, a string to be accepted, it must start with **1** "to avoid the trap state p" and contain an even number of **1s** - including the first one - to pass the toggle between q and r state.

QUESTION 8



(a)

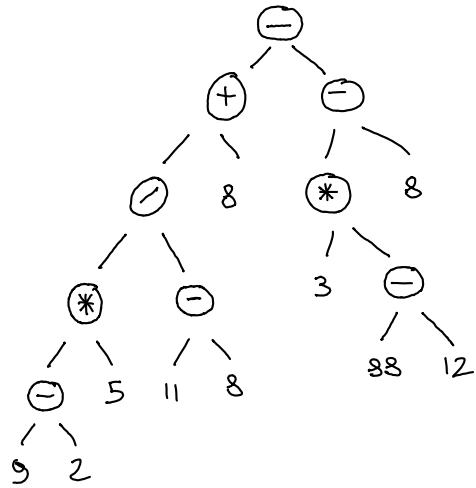
i) Tree Leaf nodes are those once childless. From the diagram, the leaf nodes are: **SystemExit, KeyboardInterrupt, ValueError, ZerodivisionError, IndexError, KeyError.**

ii) The height of a tree is the number of edges on the longest path from the root to a leaf.
-Level 0: BaseException.
-Level 1: SystemExit, Exception, KeyboardInterrupt.
-Level 2: ValueError, LookupError, ArithmeticError.
-Level 3: IndexError, KeyError, ZeroDivisionError.
Thus, the longest root-to-leaf path has **3 edges**. (BaseException -> Exception -> LookupError -> IndexError).

iii) Siblings share the same parent. In this case, ArithmeticError's parent is Exception, whose other children are: **LookupError** and **ValueError**.

b

i)



ii) Counting the root as **Level 1**, the deepest chain goes:
-Level 1: the root "-".
-Level 2: child "+".
-Level 3: child "/".
-Level 4: child "*".
-Level 5: child "-".
-Level 6: leaf "9" or "2".
There are **6 levels** in total.

iii) In a full binary tree, level k can contain up to 2^(k-1) nodes - if the root is level 1. At level 5, the maximum number of nodes is $\mathbf{2^{5-1} = 2^4 = 16.}$

iv) A general multiway tree provides operations such as root(), parent(node), and children(node). A binary-tree ADT usually extends these methods that exploit the binary structure, for instance: **left(node), right(node), hasLeft(node)/hasRight(node), setLeft(node, subtree)/setRight(node, subtree).** These provide direct access to the child links in a tree, rather than returning children as generic lists.

QUESTION 9

i) It is easier to insert and delete elements in a linked-list-based structure than in an array-based structure because: Linked lists only require changing a few pointers when adding or removing elements without shifting the rest of the elements - **inserting or deliting given a reference to the node is therefore O(1) in a linked list.** Whereas arrays, they require shifting elements to maintain continuity of storage whenever an insider inserts or deletes in the middle, which is O(n) in the worst case. This overhead makes insertions and deletions more expensive for array-based structures.

ii) In python, these data structures each fill a distict role, particularly when considerting memory allocation strategies and the way they store elements. **A string**, is an immutable sequence of characters. Once a string is generated, any apparent alteration under the hood actually creates a new string, rather than updating the existing one. Such a behavior has a direct impact on memory allocation. Because strings cannot change in size, Python can store them in a continuous block of memory and easily calculate their required space at creation time. For certain commonly used or short strings, Python can also apply optimizations, such as interning, to reduce memory duplication across the program. This immutability is particularly valuable when strings are used as key in dictionaries or when they are shared among multiple parts of an application.

In contrast, **a list** is a mutable, dynamic array of references to Python objects. A list can contain elements of any type, and it supports operations like appending, removing, or replacing elements in place. To facilitate efficient append operations, Python's list implementation uses over-allocation: it maintains extra space beyond the current number of elements. This strategy allows the list to accomodate new items without reallocating memory every time it grows, optimizing average insertion time. However, inserting or removing from the middle of a list necessitates shifting elements, which can be expensive in proportion to the size of the list. Hence, while lists offer flexibility and powerful slice operations, they demand more careful performance considerations for frequent insertions or deletions in non-terminal positions.

Finally, **a tuple** shares the immutability of a string but mirrors the list's structure of holding references to objects. Because it is immutable, Python can allocate exactly the space needed for the tuple's contents. Tuples often serve as lightweight, fixed-size containers for data that will not change - such as records or return values from functions.

(b)

i)

OPERATION	RETURN VALUE	STACK
S.PUSH(5)	5	[5]
S.PUSH(3)	3	[5, 3]
S.PUSH(22)	22	[5, 3, 22]
PRINT(S.TOP())	22	[5, 3, 22]
PRINT(S.POP())	22	[5, 3]
S.PUSH(7)	7	[5, 3, 7]
PRINT(S.POP())	7	[5, 3]
PRINT(S.POP())	3	[5]

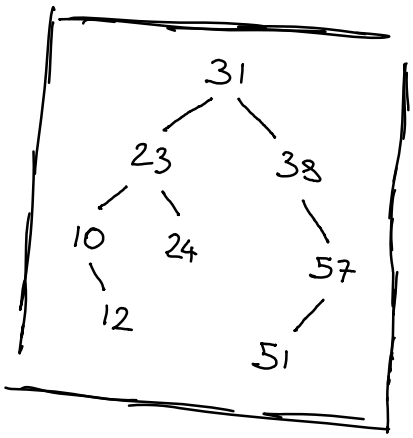
ii)

OPERATION	VALUE	QUEUE
Q.ENQUEUE(15)	15	[15]
Q.ENQUEUE(31)	31	[15, 31]
Q.ENQUEUE(4)	4	[15, 31, 4]
PRINT(Q.DEQUEUE())	15	[31, 4]
Q.ENQUEUE(11)	11	[31, 4, 11]
PRINT(Q.DEQUEUE())	31	[4, 11]
PRINT(Q.DEQUEUE())	4	[11]
PRINT(Q.DEQUEUE())	11	—

QUESTION 10

(a)

i)



Each new key is compared to the current node and goes left if smaller or right if larger, moving down until an empty spot is found.

ii) **Pre-order traversal** (Root, Left, Right)
-Visit the root (31).

-Recur left visiting (23), recur left visiting (10), recur left none then right (12).
-Recur right visiting (24).

-Recur right visiting (38), recur left none then right (57), recur left visiting (51), recur right none.

The pre-order sequence is: **31, 23, 10, 12, 24, 38, 57, 51.**

iii) **Post-order traversal** (Left, Right, Root)
-Recur left none then right (12), visit (10), recur right visiting (24) then visit (23).

-Recur left visiting (51), recur right none then visit (57) again, visit 38.

-Visit the root (31)

The post-order sequence is: **12, 10, 24, 23, 51, 57, 38, 31.**

iv) **In-order traversal** (Left, Root, Right)
-At (10), recur left none then revisit (10), recur right visiting (12), visit (23), recur right visiting (24).

-Visit the root (31).

-At (38), recur left none then revisit (38), recur right visiting (57), recur left visiting (51), visit (57), recur right none.

The in-order sequence is: **10, 12, 23, 24, 31, 38, 51, 57.**

Binary search trees often arise in scenarios requiring fast lookup or structured storage of data. A common real-world example is maintaining a sorted collection of records - a database index for quickly locating a customer record by their ID. By building a BST keyed on the ID field, one can determine whether an incoming search ID is in the left or right subtree at each step. This allows lookups, insertions, and deletions to operate in O(log n) time on average.

Suppose a small retail company wants to keep track of product inventory in real time and quickly retrieve products by their unique ID. If the IDs are stored in a BST, new products can be inserted as they arrive, old products removed when discontinued, and lookups performed to find product details for sales transactions. If the data was stored in a simple list, searches could degrade to O(n); however, with a BST, searches scale more gracefully. Thus, building and operating with a binary search tree helps ensure that the application remains responsive even as the number of product grows.