



GANDHI SCHOOL OF ENGINEERING

Brahmapur, Odisha

E-Notes

Programming in C and Data Structure

Complete E-Notes

B.Tech 1st Year · Semester 1 · Circuit Branch

Academic Year 2026-27



How to use these notes

These notes follow the official 2026-27 syllabus for Programming in C and Data Structure (3-0-0, 3 credits, 40 contact hours, 100 university marks + 50 internal).

Pre-requisite: Fundamentals of Computers. Each module has these parts:

- **Theory** with syntax, rules and diagrams.
- **Complete, tested C programs** with their output. Every full program in these notes compiles with gcc.
- **Solved problems:** "write a program" questions and "predict the output" questions with explanations.
- **Practice problems with solutions.**
- **MCQs** with answers and reasons.

At the end: 2-mark questions with answers, a long-question bank, a model paper and a quick-reference sheet. The Programming Lab has a separate lab manual containing all 30 lab programs.

Module	Topic	Hours
I	Fundamentals of C: algorithms and flowcharts, structure of a C program, tokens, data types, operators, precedence, input/output, control structures	10
II	Functions, recursion, arrays (1-D, 2-D), matrix operations, strings, passing arrays to functions, structures and unions	9
III	Pointers and dynamic memory: pointer arithmetic, call by value vs reference, pointer to pointer, pointers with structures and arrays, malloc, calloc, realloc, free	8
IV	Data structures: linked list (creation, insertion, deletion), stack and its applications (infix to postfix, postfix evaluation), queue (linear and circular)	7
V	Trees, sorting and searching: binary tree, binary search tree, bubble sort, quick sort, linear search, binary search	6

How to compile and run a C program

On Linux (or the lab computers): save the program as prog.c, then

```
gcc prog.c -o prog      (compile; add -lm if you use math.h functions)
./prog                  (run)
```

On Windows, use Code::Blocks, Dev-C++ or VS Code with MinGW; on a phone, apps like "Coding C" or online compilers can be used for practice.



Reference books (from syllabus)

- Byron Gottfried, Schaum's Outline of Programming with C, McGraw-Hill.
- Pradip Dey and Manas Ghosh, Programming in C, Oxford.
- Yashavant Kanetkar, Let Us C, BPB.
- B.W. Kernighan and D.M. Ritchie, The C Programming Language, Prentice Hall.
- A.K. Rath and A.K. Jagadev, Data Structures Using C, Scitech.



Contents

How to use these notes	2
Programming in C and Data Structure	7
Module I: Fundamentals of C	7
1.1 Problem-solving process	7
1.2 Algorithms	7
1.3 Flowcharts	7
1.4 C as a middle-level language	8
1.5 Structure of a C program	9
1.6 Character set, tokens, keywords and identifiers	9
1.7 Data types	10
1.8 Constants and variables	10
1.9 Statements, input and output	11
1.10 Operators and expressions	12
1.11 Precedence and associativity of operators	12
1.12 Control structures: decision making	13
1.13 Loops	14
1.14 Complete programs for Module I	15
1.15 Solved problems: predict the output (Module I)	17
1.16 Solved problems: write the program (Module I)	19
1.17 Practice problems with solutions: Module I	21
1.18 MCQs: Module I	22
Module II: Functions, Arrays, Structures and Unions	26
2.1 Functions	26
2.2 Recursion	26
2.3 Arrays	27
2.4 Matrix operations	27
2.5 Strings	28
2.6 Passing arrays to functions	29
2.7 Structures	29
2.8 Unions	30
2.9 Common mistakes to avoid in Module II	31
2.10 Solved problems: Module II	31
2.11 Practice problems with solutions: Module II	35
2.12 MCQs: Module II	36
Module III: Pointers and Dynamic Memory Allocation	40
3.1 Pointer basics	40
3.2 Pointer arithmetic	40



3.3 Pointers and arrays	41
3.4 Parameter passing using pointers: call by value vs call by reference	41
3.5 Pointer to pointer	42
3.6 Pointers to structures and functions	42
3.7 Static vs dynamic memory	42
3.8 Dynamic memory allocation functions (stdlib.h)	43
3.9 Common mistakes with pointers	44
3.10 Solved problems: Module III	44
3.11 Practice problems with solutions: Module III	47
3.12 MCQs: Module III	48
Module IV: Data Structures (Linked List, Stack, Queue)	52
4.1 Introduction to data structures	52
4.2 Linked list	52
4.3 Stack	55
4.4 Arithmetic expression notations	56
4.5 Infix to postfix conversion (algorithm)	57
4.6 Evaluation of a postfix expression	58
4.7 Queue	59
4.8 Common mistakes to avoid in Module IV	61
4.9 Solved problems: Module IV	61
4.10 Practice problems with solutions: Module IV	63
4.11 MCQs: Module IV	64
Module V: Trees, Sorting and Searching	68
5.1 Trees: terminology	68
5.2 Binary tree	68
5.3 Tree traversals	69
5.4 Binary search tree (BST)	69
5.5 Searching	72
5.6 Sorting	73
5.7 Bubble sort	73
5.8 Quick sort	74
5.9 Common mistakes to avoid in Module V	75
5.10 Solved problems: Module V	75
5.11 Practice problems with solutions: Module V	77
5.12 MCQs: Module V	78
Additional Solved Programs (All Modules)	82
Find the Error (Debugging Exercises)	90
Short-Answer Questions (2 marks each) with Answers	92



MCQ Practice Set 2: Predict the Output	97
Long-Answer Question Bank (with section references)	100
Model Question Paper with Solutions	101
Quick Reference Sheet (Last-Minute Revision)	104
Glossary and ASCII Table	106
Exam Preparation Checklist	109



Programming in C and Data Structure

Subject: Programming in C and Data Structure · Semester 1 (Circuit branch) · 3-0-0 · 3 credits · 40 hours ·
University 100 + Internal 50

Module I: Fundamentals of C

1.1 Problem-solving process

Solving a problem with a computer follows these steps:

- **Problem definition:** understand exactly what is given (input) and what is required (output).
- **Analysis:** identify the formulas, conditions and constraints.
- **Algorithm design:** write a step-by-step method.
- **Flowchart / pseudocode:** represent the algorithm graphically or in structured English.
- **Coding:** translate the algorithm into a programming language (C).
- **Compilation and debugging:** remove syntax errors (reported by the compiler) and logical errors (wrong output).
- **Testing:** run the program with different inputs, including boundary cases.
- **Documentation and maintenance.**

1.2 Algorithms

An **algorithm** is a finite sequence of well-defined, unambiguous steps that solves a problem in a finite amount of time.

Characteristics of a good algorithm:

- **Input:** zero or more well-defined inputs.
- **Output:** at least one output.
- **Definiteness:** each step is clear and unambiguous.
- **Finiteness:** it terminates after a finite number of steps.
- **Effectiveness:** each step is basic enough to be carried out exactly.

Example: algorithm to find the largest of three numbers.

- Step 1: Start.
- Step 2: Read A, B, C.
- Step 3: If $A > B$ and $A > C$, then $MAX = A$.
- Step 4: Else if $B > C$, then $MAX = B$; else $MAX = C$.
- Step 5: Print MAX.
- Step 6: Stop.

Example: algorithm to find the sum of the first N natural numbers.

- Step 1: Start. Step 2: Read N. Step 3: $SUM = 0$, $I = 1$.
- Step 4: While $I \leq N$: $SUM = SUM + I$; $I = I + 1$.
- Step 5: Print SUM. Step 6: Stop.

1.3 Flowcharts

A **flowchart** is a diagram of an algorithm using standard symbols connected by arrows (flow lines).

Symbol	Name	Use
Oval (rounded rectangle)	Terminal	Start / Stop
Parallelogram	Input/Output	Read or print data
Rectangle	Process	Calculations and assignments
Diamond	Decision	Condition with Yes/No (True/False) branches
Circle	Connector	Joins parts of a flowchart (same page)
Arrow	Flow line	Direction of flow
Rectangle with double side lines	Predefined process	Call to a function/sub-routine

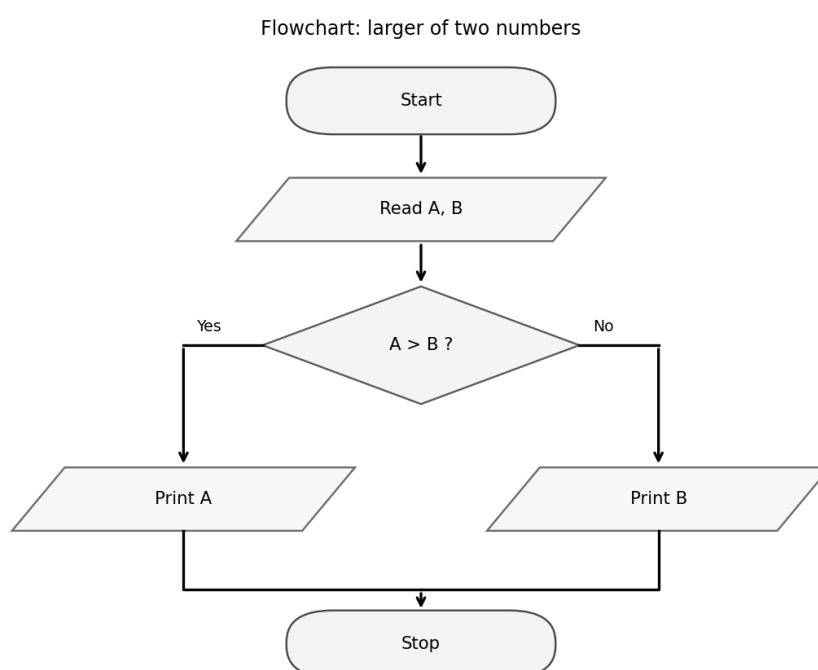


Figure 1.1: Flowchart to find the larger of two numbers.

Advantages: easy to understand and communicate logic; helps in debugging and documentation. **Limitations:** difficult to draw for large programs; modifications need redrawing.

Pseudocode is an informal, English-like description of an algorithm using programming constructs (IF...THEN...ELSE, WHILE, FOR), without strict syntax.

1.4 C as a middle-level language

C was developed by **Dennis Ritchie** at **AT&T Bell Laboratories** in **1972**, mainly to write the UNIX operating system. It evolved from the languages BCPL and B.

C is called a **middle-level language** because it combines features of both:

- **High-level features:** structured programming, functions, data types, portability, readability.
- **Low-level features:** direct memory access through pointers, bitwise operators, ability to write system software (operating systems, device drivers, compilers, embedded

programs).

Features of C: simple and efficient; portable (a program written for one machine runs on another with little change); structured (programs divided into functions); rich set of operators and library functions; supports dynamic memory allocation; extensible; fast execution; widely used in embedded systems and operating systems.

1.5 Structure of a C program

```
/* Documentation section: program name, author, purpose */
#include <stdio.h>           /* Link section: header files */
#define PI 3.14159          /* Definition section: symbolic constants */
int count = 0;              /* Global declaration section */
float area(float r);        /* Function prototype */

int main(void)              /* main() function section */
{
    float r = 2.0;          /* declaration part */
    printf("Area = %.2f\n", area(r)); /* executable part */
    return 0;
}

float area(float r)         /* Sub-program section: user-defined functions */
{
    return PI * r * r;
}
```

- **Documentation section:** comments (`/* ... */` or `// ...`) describing the program. Ignored by the compiler.
- **Link (preprocessor) section:** `#include` directives that bring in header files (`stdio.h` for `printf/scanf`, `math.h` for `sqrt/pow`, `string.h` for `strlen/strcpy`, `stdlib.h` for `malloc/exit`).
- **Definition section:** `#define` for symbolic constants and macros.
- **Global declarations:** variables and function prototypes visible to all functions.
- **main() function:** every C program must have exactly one `main()`; execution begins here.
- **Sub-program section:** user-defined functions.

Compilation process: source code (.c) → **preprocessor** (handles `#include`, `#define`; produces expanded source) → **compiler** (translates to assembly) → **assembler** (produces object code .o) → **linker** (combines object code with library functions) → executable (a.out or .exe) → **loader** loads it into memory for execution.

1.6 Character set, tokens, keywords and identifiers

Character set: letters (A-Z, a-z), digits (0-9), special characters (+ - * / % = < > () { } [] ; , . : ' " # & | ^ ~ ! ? _ \), and white spaces (blank, tab, newline).

Tokens are the smallest individual units of a program: keywords, identifiers, constants, strings, operators and special symbols.

Keywords are reserved words with a fixed meaning; they cannot be used as identifiers. C (C89) has 32 keywords:

Keywords of C
auto, break, case, char, const, continue, default, do
double, else, enum, extern, float, for, goto, if

Keywords of C

int, long, register, return, short, signed, sizeof, static

struct, switch, typedef, union, unsigned, void, volatile, while

Identifiers are names given to variables, functions, arrays, etc. Rules:

- Can contain letters, digits and underscore only.
- Must begin with a letter or underscore (not a digit).
- Case-sensitive (Sum, sum and SUM are different).
- Cannot be a keyword; no spaces or special characters.

Valid: total, _count, marks1, gross_salary. Invalid: 1st (starts with digit), gross salary (space), int (keyword), rate% (special character).

1.7 Data types

Data type	Typical size (bytes, 32/64-bit gcc)	Range (typical)	Format specifier
char	1	−128 to 127	%c (character), %d (value)
unsigned char	1	0 to 255	%c, %u
short int	2	−32768 to 32767	%hd
int	4	−2,147,483,648 to 2,147,483,647	%d or %i
unsigned int	4	0 to 4,294,967,295	%u
long int	4 or 8	at least $\pm 2.1 \times 10^9$	%ld
long long int	8	about $\pm 9.2 \times 10^{18}$	%lld
float	4	$\pm 3.4 \times 10^{38}$, about 6–7 significant digits	%f, %e, %g
double	8	$\pm 1.7 \times 10^{308}$, about 15 significant digits	%lf (scanf), %f (printf)
long double	10–16	higher precision	%Lf
void	—	no value	—

Sizes depend on the compiler and machine; use the **sizeof** operator to find them. (Old 16-bit Turbo C used 2-byte int with range −32768 to 32767; many older books give that value.)

Derived data types: arrays, pointers, functions. **User-defined data types:** structures, unions, enumerations (enum), typedef names.

Type qualifiers and modifiers: signed, unsigned, short, long, const (value cannot change), volatile.

1.8 Constants and variables

Constants are fixed values that do not change during execution.

- **Integer constants:** decimal (25, −17), octal (starting with 0: 031), hexadecimal (starting with 0x: 0x1F).
- **Real (floating) constants:** 3.14, −0.005, 2.5e3 (= 2500), 1.2E−4.

- **Character constants:** a single character in single quotes: 'A', '5', '\n'. Each has an integer (ASCII) value: 'A' = 65, 'a' = 97, '0' = 48.
- **String constants:** characters in double quotes: "Hello". A string is stored with a terminating null character '\0', so "Hello" occupies 6 bytes.
- **Escape sequences:** \n newline, \t tab, \\ backslash, \' single quote, \" double quote, \0 null, \a alert (bell), \b backspace, \r carriage return.
- **Symbolic constants:** #define PI 3.14159 (no semicolon, no memory), or const float pi = 3.14159; (typed, occupies memory).

Variables are named memory locations whose values can change. A variable must be **declared** before use: data_type name; e.g. int age; float salary = 25000.5; char grade = 'A';

Scope and storage classes.

Storage class	Where stored	Default initial value	Scope	Lifetime
auto (default for local)	Memory (stack)	Garbage	Within the block	Until the block ends
register	CPU register (if available)	Garbage	Within the block	Until the block ends
static	Memory	Zero	Within the block (or file)	Entire program run (value persists between calls)
extern (global)	Memory	Zero	Whole program (all files)	Entire program run

1.9 Statements, input and output

A **statement** is an instruction ending with a semicolon. Types: expression statements (`x = y + 1;`), compound statements or blocks (`{ ... }`), control statements (`if`, `for`, `while`, `switch`).

Formatted output: printf(). `printf("control string", arg1, arg2, ...);` The control string contains text and format specifiers (`%d`, `%f`, `%c`, `%s`, `%u`, `%ld`, `%x`, `%o`, `%e`, `%g`).

- Field width and precision: `%5d` prints an integer in 5 columns (right-justified); `%-5d` left-justified; `%8.2f` prints a float in 8 columns with 2 decimal places; `%.3f` three decimals.

Formatted input: scanf(). `scanf("control string", &var1, &var2, ...);` The address-of operator `&` is required for ordinary variables (not for strings, since an array name is already an address).

Unformatted character I/O: `getchar()` reads one character; `putchar(c)` prints one; `gets()/fgets()` read a line (`fgets(str, size, stdin)` is safe; `gets()` is unsafe and removed from modern C); `puts(str)` prints a string with a newline.

```
#include <stdio.h>
int main(void)
{
    int age; float height; char grade;
    printf("Enter age, height and grade: ");
    scanf("%d %f %c", &age, &height, &grade);
    printf("Age = %d\nHeight = %.2f m\nGrade = %c\n", age, height, grade);
    return 0;
}
```

Input: 18 1.72 A → Output: Age = 18, Height = 1.72 m, Grade = A (each on a new line).

1.10 Operators and expressions

Category	Operators	Notes
Arithmetic	+ - * / %	% (modulus) works only on integers; integer division truncates: 7/2 = 3
Relational	< <= > >= == !=	Result is 1 (true) or 0 (false)
Logical	&& (AND), (OR), ! (NOT)	Short-circuit evaluation: in a && b, b is not evaluated if a is false
Assignment	= += -= *= /= %=	a += 5 means a = a + 5
Increment / decrement	++ --	Prefix (++a: increment, then use) and postfix (a++: use, then increment)
Conditional (ternary)	? :	max = (a > b) ? a : b;
Bitwise	& (AND), (OR), ^ (XOR), ~ (NOT), << (left shift), >> (right shift)	a << 1 multiplies by 2; a >> 1 divides by 2
Special	sizeof, comma (,), & (address), * (dereference), . and -> (member access)	sizeof(int) gives the size in bytes

(Here || stands for the vertical-bar character used by C for OR and bitwise OR.)

Type conversion.

- **Implicit (automatic):** in mixed expressions the lower type is promoted to the higher type (char → int → long → float → double). Example: 5/2 = 2 but 5/2.0 = 2.5.
- **Explicit (type casting):** (type) expression. Example: (float)5/2 = 2.5; (int)3.99 = 3.

Prefix vs postfix example. int a = 5, b; b = a++; gives b = 5, a = 6. b = ++a; gives a = 7, b = 7.

1.11 Precedence and associativity of operators

When an expression contains several operators, **precedence** decides which is evaluated first, and **associativity** decides the order among operators of equal precedence.

Precedence (high to low)	Operators	Associativity
1	() [] -> . (postfix ++ --)	Left to right
2	Unary: ! ~ ++ -- + - * & (type) sizeof	Right to left
3	* / %	Left to right



Precedence (high to low)	Operators	Associativity
4	+ -	Left to right
5	<< >>	Left to right
6	< <= > >=	Left to right
7	== !=	Left to right
8	& (bitwise AND)	Left to right
9	^ (bitwise XOR)	Left to right
10	(bitwise OR)	Left to right
11	&& (logical AND)	Left to right
12	(logical OR)	Left to right
13	?: (conditional)	Right to left
14	= += -= *= /= %= etc.	Right to left
15	, (comma)	Left to right

Examples.

- $2 + 3 * 4 = 14$ (multiplication first).
- $10 - 4 - 3 = 3$ (left to right: $(10 - 4) - 3$).
- $a = b = c = 5$ assigns 5 to all (right to left).
- $8 / 2 * 4 = 16$ (left to right: $(8/2) * 4$).
- $5 > 3 \ \&\& \ 2 > 4 = 0; \ !(5 > 3) = 0; \ 10 \% 3 * 2 = 2.$

Note: Use parentheses to make the intended order clear; it costs nothing and avoids precedence errors.

1.12 Control structures: decision making

Programs normally execute statements in sequence. Control structures change this order: **selection** (if, if-else, else-if ladder, nested if, switch), **iteration** (for, while, do-while) and **jumps** (break, continue, goto, return).

(a) if statement. if (condition) statement; — the statement runs only when the condition is true (non-zero).

(b) if-else statement.

```
if (n % 2 == 0)
    printf("Even\n");
else
    printf("Odd\n");
```

(c) else-if ladder (multi-way decision): conditions are tested from the top; the first true one is executed, and the rest are skipped.

```
if (marks >= 90)      grade = 'O';
else if (marks >= 80) grade = 'E';
else if (marks >= 70) grade = 'A';
else if (marks >= 60) grade = 'B';
else if (marks >= 50) grade = 'C';
else if (marks >= 40) grade = 'D';
else                  grade = 'F';
```

(d) Nested if: an if inside another if. An else always pairs with the nearest unmatched if (**dangling else**); use braces to avoid confusion.

(e) switch statement. Selects one of many alternatives based on the value of an integer or character expression.

```
switch (day) {
    case 1: printf("Monday\n"); break;
    case 2: printf("Tuesday\n"); break;
    /* ... */
    case 7: printf("Sunday\n"); break;
    default: printf("Invalid day\n");
}
```

Rules: the expression must be integral (int or char, not float or string); case labels must be unique constants; break exits the switch (without break, execution **falls through** into the next case); default is optional and runs when no case matches.

if-else ladder	switch
Can test any condition (ranges, float, logical expressions)	Tests only equality with integer/character constants
Evaluated one by one from the top	Jumps directly to the matching case (often faster)
Suitable for ranges (marks ≥ 60)	Suitable for menus and fixed values

1.13 Loops

(a) for loop: used when the number of iterations is known.

```
for (initialisation; condition; update)
    body;
```

Example: for (i = 1; i <= 10; i++) sum += i; Execution order: initialisation once → test condition → body → update → test condition → ... → exit when the condition is false.

(b) while loop: entry-controlled; the condition is tested before each iteration, so the body may execute zero times.

```
while (n > 0) {
    digit = n % 10;
    sum += digit;
    n /= 10;
}
```

(c) do-while loop: exit-controlled; the body executes at least once, and the condition is tested after each iteration. Note the semicolon after while(...).

```
do {
    printf("Enter a positive number: ");
    scanf("%d", &n);
} while (n <= 0);
```

while	do-while
Entry-controlled (test first)	Exit-controlled (test last)
Body may execute zero times	Body executes at least once
while (cond) { ... }	do { ... } while (cond);

Nested loops: a loop inside another loop; used for patterns, matrices and tables. The inner loop completes all its iterations for each iteration of the outer loop.

Infinite loops: for(;;) { ... } or while (1) { ... }, exited with break.

(d) Jump statements.

- **break:** exits the innermost loop or switch immediately.
- **continue:** skips the rest of the current iteration and goes to the next iteration (in a for loop, the update is executed).
- **goto label;** transfers control to a labelled statement. Avoid it; it makes programs hard to follow (spaghetti code).
- **return:** exits from a function, optionally returning a value.

Example: difference between break and continue.

```
for (i = 1; i <= 5; i++) {           /* prints 1 2 */
    if (i == 3) break;
    printf("%d ", i);
}
for (i = 1; i <= 5; i++) {           /* prints 1 2 4 5 */
    if (i == 3) continue;
    printf("%d ", i);
}
```

1.14 Complete programs for Module I

Program 1.1: Largest of three numbers.

```
#include <stdio.h>
int main(void)
{
    int a, b, c, max;
    printf("Enter three numbers: ");
    scanf("%d %d %d", &a, &b, &c);
    max = a;
    if (b > max) max = b;
    if (c > max) max = c;
    printf("Largest = %d\n", max);
    return 0;
}
```

Program 1.2: Roots of a quadratic equation $ax^2 + bx + c = 0$.

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    double a, b, c, d, r1, r2;
    printf("Enter a, b, c: ");
    scanf("%lf %lf %lf", &a, &b, &c);
    d = b * b - 4 * a * c;
    if (d > 0) {
        r1 = (-b + sqrt(d)) / (2 * a);
        r2 = (-b - sqrt(d)) / (2 * a);
        printf("Real and distinct roots: %.2f, %.2f\n", r1, r2);
    } else if (d == 0) {
        printf("Real and equal roots: %.2f\n", -b / (2 * a));
    } else {
        printf("Complex roots: %.2f +/- %.2fi\n", -b / (2 * a), sqrt(-d) / (2 * a));
    }
    return 0;
}
```

Compile with gcc prog.c -lm. Sample: a = 1, b = -5, c = 6 → roots 3.00, 2.00.

Program 1.3: Sum of digits and reverse of a number.

```
#include <stdio.h>
int main(void)
{
    int n, rev = 0, sum = 0, d;
    printf("Enter a number: ");
    scanf("%d", &n);
    while (n > 0) {
        d = n % 10;
        sum += d;
        rev = rev * 10 + d;
        n /= 10;
    }
    printf("Sum of digits = %d, Reverse = %d\n", sum, rev);
    return 0;
}
```

Input 1234 → Sum of digits = 10, Reverse = 4321.

Program 1.4: Factorial and Fibonacci series using loops.

```
#include <stdio.h>
int main(void)
{
    int n, i;
    long long fact = 1, a = 0, b = 1, t;
    printf("Enter n: ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++) fact *= i;
    printf("%d! = %lld\n", n, fact);
    for (i = 1; i <= n; i++) {
        printf("%lld ", a);
        t = a + b; a = b; b = t;
    }
    printf("\n");
    return 0;
}
```

Input 6 → 6! = 720; Fibonacci: 0 1 1 2 3 5.

Program 1.5: Check whether a number is prime.

```
#include <stdio.h>
int main(void)
{
    int n, i, prime = 1;
    printf("Enter a number: ");
    scanf("%d", &n);
    if (n < 2) prime = 0;
    for (i = 2; i * i <= n && prime; i++)
        if (n % i == 0) prime = 0;
    printf("%d is %s\n", n, prime ? "prime" : "not prime");
    return 0;
}
```

Only divisors up to $\sqrt{n}$ need to be checked, because if $n = a \times b$ then one of a, b is at most $\sqrt{n}$.

Program 1.6: Menu-driven calculator using switch.

```
#include <stdio.h>
int main(void)
{
    char op; double x, y;
    printf("Enter expression (e.g. 4 * 5): ");
    scanf("%lf %c %lf", &x, &op, &y);
    switch (op) {
        case '+': printf("%.2f\n", x + y); break;
        case '-': printf("%.2f\n", x - y); break;
        case '*': printf("%.2f\n", x * y); break;
        case '/':
            if (y != 0) printf("%.2f\n", x / y);
            else printf("Division by zero!\n");
            break;
        default: printf("Invalid operator\n");
    }
    return 0;
}
```

1.15 Solved problems: predict the output (Module I)

Problem 1. What is the output? `int a = 10, b = 3; printf("%d %d %.2f", a / b, a % b, (float)a / b);`

Integer division $10/3 = 3$; remainder $10 \% 3 = 1$; with a cast, $10.0/3 = 3.33$.

Answer: 3 1 3.33

Problem 2. What is printed? `int x = 5; printf("%d %d %d", x++, x, ++x);`

The order of evaluation of function arguments is unspecified in C, and modifying x twice without a sequence point is **undefined behaviour**.

Answer: The output is compiler-dependent (undefined behaviour). Never write such code in exams or programs; if asked, state this.

Problem 3. `int a = 5, b; b = a++ + 10; printf("%d %d", a, b);`

Postfix: the old value 5 is used, then a becomes 6. $b = 5 + 10 = 15$.



Answer: 6 15

Problem 4. `int a = 5, b; b = ++a * 2; printf("%d %d", a, b);`

Prefix: a becomes 6 first; b = 12.

Answer: 6 12

Problem 5. `printf("%d", 2 + 3 * 4 - 8 / 2 % 3);`

* / % first, left to right: $3 * 4 = 12$; $8 / 2 = 4$; $4 \% 3 = 1$. Then $2 + 12 - 1 = 13$.

Answer: 13

Problem 6. `int i = 0; while (i < 5) i += 2; printf("%d", i);`

$i = 0 \rightarrow 2 \rightarrow 4 \rightarrow 6$; the loop stops when $i = 6$.

Answer: 6

Problem 7. `int i; for (i = 0; i < 10; i++); printf("%d", i);`

The semicolon after for makes the body empty; the loop just runs until $i = 10$, and then printf executes once.

Answer: 10

Problem 8. `int x = 7; if (x = 0) printf("zero"); else printf("non-zero");`

$x = 0$ is an assignment, not a comparison. Its value is 0 (false), so the else part executes.

Answer: non-zero (a common bug: use == for comparison).

Problem 9. `char c = 'A'; printf("%c %d %c", c, c, c + 2);`

'A' has ASCII value 65; $'A' + 2 = 67 = 'C'$.

Answer: A 65 C

Problem 10. `int a = 10, b = 20; printf("%d", a > b ? a : b);`

$10 > 20$ is false, so the value is b.

Answer: 20

Problem 11. `int n = 3; switch (n) { case 1: printf("one "); case 3: printf("three "); case 4: printf("four "); default: printf("other"); }`

No break statements, so after matching case 3, execution falls through all later cases.

Answer: three four other

Problem 12. `printf("%d %d", 5 & 3, 5 | 3); and printf("%d %d", 5 ^ 3, 5 << 2);`

$5 = 101$, $3 = 011$. AND = $001 = 1$; OR = $111 = 7$; XOR = $110 = 6$; $5 << 2 = 10100 = 20$.

Answer: 1 7 and 6 20



Problem 13. `int i, j; for (i = 1; i <= 3; i++) { for (j = 1; j <= i; j++) printf("*"); printf("\n"); }`

Row i prints i stars.

Answer: * / * on three lines (a right-angled triangle).

Problem 14. `int x = 10; x += x -= 5; printf("%d", x);`

Assignment is right to left, but x is modified twice without a sequence point:
undefined behaviour in C. (Most compilers give 10.)

Answer: Undefined behaviour; avoid such expressions.

Problem 15. `printf("%d", sizeof(char) + sizeof(int) + sizeof(double));` (typical gcc)

1 + 4 + 8.

Answer: 13 (on a typical 32/64-bit system).

Problem 16. `float f = 5 / 2; printf("%.1f", f);`

5/2 is integer division = 2, then converted to float 2.0.

Answer: 2.0 (write 5.0/2 to get 2.5).

1.16 Solved problems: write the program (Module I)

Problem 1. Write a program to find the simple interest and compound interest for given principal, rate and time.

```
#include <stdio.h>
#include <math.h>
int main(void)
{
    double p, r, t, si, ci;
    printf("Enter principal, rate (%%), time (years): ");
    scanf("%lf %lf %lf", &p, &r, &t);
    si = p * r * t / 100;
    ci = p * pow(1 + r / 100, t) - p;
    printf("Simple interest   = %.2f\n", si);
    printf("Compound interest = %.2f\n", ci);
    return 0;
}
```

ANS: For p = 10000, r = 10, t = 2: SI = 2000.00, CI = 2100.00.

Problem 2. Write a program to check whether a year is a leap year.

A year is a leap year if it is divisible by 4 but not by 100, or if it is divisible by 400.

```
#include <stdio.h>
int main(void)
{
    int y;
    printf("Enter year: ");
    scanf("%d", &y);
    if ((y % 4 == 0 && y % 100 != 0) || y % 400 == 0)
        printf("%d is a leap year\n", y);
    else
        printf("%d is not a leap year\n", y);
    return 0;
}
```

ANS: 2024 → leap year; 1900 → not a leap year; 2000 → leap year.

Problem 3. Write a program to check whether a number is an Armstrong number (e.g. $153 = 1^3 + 5^3 + 3^3$).

```
#include <stdio.h>
int main(void)
{
    int n, t, d, digits = 0, sum = 0, i, p;
    printf("Enter a number: ");
    scanf("%d", &n);
    for (t = n; t > 0; t /= 10) digits++;
    for (t = n; t > 0; t /= 10) {
        d = t % 10;
        for (p = 1, i = 0; i < digits; i++) p *= d;
        sum += p;
    }
    printf("%d is %san Armstrong number\n", n, sum == n ? "" : "not ");
    return 0;
}
```

ANS: 153 → Armstrong number; 123 → not an Armstrong number.

Problem 4. Write a program to find the GCD and LCM of two numbers (Euclid's algorithm).

```
#include <stdio.h>
int main(void)
{
    int a, b, x, y, r;
    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);
    x = a; y = b;
    while (y != 0) { r = x % y; x = y; y = r; }
    printf("GCD = %d, LCM = %d\n", x, a / x * b);
    return 0;
}
```

ANS: 12 and 18 → GCD = 6, LCM = 36.

Problem 5. Write a program to print the multiplication table of a number.

```
#include <stdio.h>
int main(void)
{
    int n, i;
    printf("Enter a number: ");
    scanf("%d", &n);
    for (i = 1; i <= 10; i++)
        printf("%d x %2d = %3d\n", n, i, n * i);
    return 0;
}
```

Problem 6. Write a program to convert a decimal number to binary.

```
#include <stdio.h>
int main(void)
{
    int n, bits[32], i = 0;
    printf("Enter a positive integer: ");
    scanf("%d", &n);
    if (n == 0) { printf("0\n"); return 0; }
    while (n > 0) { bits[i++] = n % 2; n /= 2; }
    for (i = i - 1; i >= 0; i--) printf("%d", bits[i]);
    printf("\n");
    return 0;
}
```

ANS: 13 → 1101.

1.17 Practice problems with solutions: Module I**Q1. Write an algorithm and draw (describe) a flowchart to find whether a number is even or odd.**

Algorithm: Start → Read N → If $N \% 2 == 0$, print "Even", else print "Odd" → Stop.

Flowchart: oval (Start) → parallelogram (Read N) → diamond ($N \% 2 == 0$?) → Yes: parallelogram (Print Even); No: parallelogram (Print Odd) → oval (Stop).

Answer: As described.

Q2. Which of these identifiers are invalid and why: `_sum`, `2ndvalue`, `float`, `total marks`, `net-pay`, `Area`?

`2ndvalue` (starts with a digit), `float` (keyword), `total marks` (space), `net-pay` (hyphen not allowed).

Answer: `_sum` and `Area` are valid; the other four are invalid.

Q3. Evaluate: `int a = 9, b = 4`; (i) `a / b` (ii) `a % b` (iii) `a / (float)b` (iv) `-a % b`.

$9/4 = 2$; $9 \% 4 = 1$; $9/4.0 = 2.25$; in C99 the remainder takes the sign of the dividend, so $-9 \% 4 = -1$.

Answer: 2, 1, 2.25, -1.

Q4. Write the output: `int i = 1`; `do { printf("%d ", i); i *= 3; } while (i < 50)`;

`i = 1, 3, 9, 27` are printed; then `i = 81` fails the test.

Answer: 1 3 9 27



Q5. Convert the following for loop into a while loop: for (i = 10; i > 0; i -= 2) printf("%d ", i);

i = 10; while (i > 0) { printf("%d ", i); i -= 2; }

Answer: Prints 10 8 6 4 2.

Q6. What is the value of x after: int x = 3 + 4 > 5 && 10 % 3 == 1;

3 + 4 = 7; 7 > 5 is 1; 10 % 3 = 1; 1 == 1 is 1; 1 && 1 = 1.

Answer: x = 1.

Q7. Write a program segment to print all even numbers from 1 to 50 using continue.

for (i = 1; i <= 50; i++) { if (i % 2 != 0) continue; printf("%d ", i); }

Answer: Prints 2 4 6 ... 50.

Q8. What is printed by printf("%5.2f|%-5d|%05d", 3.14159, 42, 42); ?

%5.2f → " 3.14" (width 5); %-5d → "42 " (left-justified); %05d → "00042" (zero-padded).

Answer: 3.14|42 |00042

Q9. Write the if-else ladder to print the type of a triangle given sides a, b, c.

if (a == b && b == c) equilateral; else if (a == b || b == c || a == c) isosceles; else scalene. (First check validity: a + b > c, b + c > a, a + c > b.)

Answer: As above.

1.18 MCQs: Module I

1. C language was developed by

- (a) Bjarne Stroustrup (b) Ken Thompson (c) James Gosling (d) Dennis Ritchie
☐ (d) — At Bell Labs, 1972.

2. C is called a middle-level language because it

- (a) combines high-level features with low-level memory access (b) is between Java and Python (c) runs only on middle-size computers (d) has medium speed
☐ (a) — Pointers, bitwise operations.

3. Every C program must contain the function

- (a) scanf() (b) start() (c) main() (d) printf()
☐ (c) — Execution begins at main.

4. Which symbol is used for decision in a flowchart?

- (a) Rectangle (b) Diamond (c) Parallelogram (d) Oval
☐ (b) — Yes/No branches.

5. Which symbol represents input/output in a flowchart?

- (a) Rectangle (b) Circle (c) Parallelogram (d) Diamond
☐ (c) — Read/Print.

6. The number of keywords in C89 is

- (a) 64 (b) 30 (c) 32 (d) 48
☐ (c) — Standard C.



- 7.** Which is a valid identifier?
(a) float (b) 3sum (c) _total (d) my-var
☐ (c) — Starts with underscore.
- 8.** Which header file is needed for printf() and scanf()?
(a) stdio.h (b) conio.h (c) string.h (d) math.h
☐ (a) — Standard I/O.
- 9.** The format specifier for a double in scanf is
(a) %c (b) %lf (c) %f (d) %d
☐ (b) — %lf for double input.
- 10.** The size of char in C is
(a) depends only on the OS (b) 4 bytes (c) 2 bytes (d) 1 byte
☐ (d) — By definition.
- 11.** The value of 7 / 2 in C is
(a) 4 (b) 3 (c) 3.0 (d) 3.5
☐ (b) — Integer division.
- 12.** The value of 7 % 2 is
(a) 3.5 (b) 0 (c) 3 (d) 1
☐ (d) — Remainder.
- 13.** The modulus operator % cannot be used with
(a) float (b) char (c) int (d) long
☐ (a) — Only integral operands.
- 14.** The result of a relational expression is
(a) a float (b) any integer (c) 1 or 0 (d) true/false keywords
☐ (c) — C uses integers.
- 15.** Which operator has the highest precedence?
(a) () (b) * (c) + (d) =
☐ (a) — Parentheses first.
- 16.** The associativity of the assignment operator is
(a) right to left (b) left to right (c) none (d) both
☐ (a) — a = b = c.
- 17.** If a = 5, the value of b after b = a++; is
(a) 4 (b) 6 (c) 0 (d) 5
☐ (d) — Postfix uses old value.
- 18.** If a = 5, the value of b after b = ++a; is
(a) 4 (b) 7 (c) 6 (d) 5
☐ (c) — Prefix increments first.
- 19.** The ternary operator is
(a) -> (b) ?? (c) ?: (d) ::
☐ (c) — Conditional operator.
- 20.** 5 << 1 equals
(a) 10 (b) 5 (c) 2 (d) 6
☐ (a) — Left shift doubles.
- 21.** 12 & 10 equals
(a) 8 (b) 2 (c) 6 (d) 14
☐ (a) — 1100 & 1010 = 1000.



- 22.** The escape sequence for a new line is
(a) \b (b) \t (c) \r (d) \n
☐ (d) — Newline.
- 23.** Which loop is guaranteed to execute at least once?
(a) while (b) for (c) do-while (d) none
☐ (c) — Exit-controlled.
- 24.** A while loop is
(a) infinite always (b) never executed (c) exit-controlled (d) entry-controlled
☐ (d) — Test before body.
- 25.** The statement used to exit a loop immediately is
(a) continue (b) exit-loop (c) break (d) goto only
☐ (c) — Terminates loop.
- 26.** continue in a loop
(a) stops the program (b) restarts main (c) skips to the next iteration (d) exits the loop
☐ (c) — Skips remaining body.
- 27.** In a switch statement, the expression must be
(a) double (b) float (c) string (d) integer or character
☐ (d) — Integral type.
- 28.** If break is missing in a switch case,
(a) default runs (b) the program stops (c) compilation error (d) control falls through to the next case
☐ (d) — Fall-through.
- 29.** The default case in switch is
(a) compulsory (b) must be first (c) must be last (d) optional
☐ (d) — Optional.
- 30.** for (;;) represents
(a) an infinite loop (b) a syntax error (c) a null loop (d) a loop that runs once
☐ (a) — No condition = true.
- 31.** What is the output of printf("%d", 10 > 5 > 2); ?
(a) error (b) 0 (c) 10 (d) 1
☐ (b) — $(10 > 5) = 1; 1 > 2 = 0$.
- 32.** The address-of operator is
(a) # (b) -> (c) * (d) &
☐ (d) — Used in scanf.
- 33.** sizeof is
(a) a macro (b) a keyword and operator (c) an operator (d) a function
☐ (b) — Compile-time operator (keyword).
- 34.** Which storage class has a default initial value of zero?
(a) register (b) static (c) auto (d) none
☐ (b) — Also extern.
- 35.** A static local variable
(a) loses its value between calls (b) cannot be changed (c) is global (d) retains its value between calls
☐ (d) — Lifetime = whole program.
- 36.** #define PI 3.14 is processed by the
(a) compiler (b) preprocessor (c) loader (d) linker
☐ (b) — Macro substitution.



37. The stage that combines object code with library code is

- (a) linker (b) loader (c) preprocessor (d) compiler

☐ (a) — Produces executable.

38. A string "HELLO" occupies how many bytes?

- (a) 4 (b) 10 (c) 6 (d) 5

☐ (c) — Including '\0'.

39. The ASCII value of 'A' is

- (a) 32 (b) 97 (c) 48 (d) 65

☐ (d) — 'a' = 97, '0' = 48.

40. Which is not a characteristic of an algorithm?

- (a) ambiguity (b) finiteness (c) effectiveness (d) definiteness

☐ (a) — Must be unambiguous.

Module II: Functions, Arrays, Structures and Unions

2.1 Functions

A **function** is a self-contained block of statements that performs a specific task. Large programs are divided into functions (modular programming).

Advantages: avoids repetition of code (reusability); makes programs easier to read, test and debug; allows team work; supports recursion.

Types.

- **Library (built-in) functions:** provided by C libraries: printf, scanf (stdio.h), sqrt, pow, sin, fabs (math.h), strlen, strcpy, strcmp, strcat (string.h), malloc, free, exit, abs, rand (stdlib.h), toupper, isdigit (ctype.h).
- **User-defined functions:** written by the programmer.

Three parts of a user-defined function.

```
int sum(int a, int b);           /* 1. Declaration (prototype) */

int main(void)
{
    int s = sum(3, 4);           /* 2. Call: 3 and 4 are actual arguments */
    return s == 7 ? 0 : 1;
}

int sum(int a, int b)           /* 3. Definition: a, b are formal parameters */
{
    return a + b;               /* return value to the caller */
}
```

- **Return type:** the type of value returned (void if nothing is returned).
- **Formal parameters** receive the values of **actual arguments**.
- **return** sends a value back and ends the function.

Categories by arguments and return value: (1) no arguments, no return value; (2) arguments, no return value; (3) no arguments, with return value; (4) arguments with return value.

Parameter passing. C passes arguments **by value**: the function receives copies, so changes to formal parameters do not affect the actual arguments. To let a function modify the caller's variables, pass their addresses (call by reference using pointers, Module III).

2.2 Recursion

A function that calls itself is **recursive**. Every recursive function needs:

- a **base case** (terminating condition) that returns without a recursive call, and
- a **recursive case** that moves towards the base case.

Example: factorial. $n! = n \times (n - 1)!$, with $0! = 1$.

```
#include <stdio.h>
long long fact(int n)
{
    if (n <= 1) return 1;          /* base case */
    return n * fact(n - 1);        /* recursive case */
}
int main(void)
{
    printf("5! = %lld\n", fact(5));
    return 0;
}
```

How it works (call stack): $\text{fact}(5) = 5 \times \text{fact}(4) = 5 \times 4 \times \text{fact}(3) = \dots = 5 \times 4 \times 3 \times 2 \times \text{fact}(1) = 120$. Each call is pushed on the stack until the base case, then the results are multiplied as the calls return.

More examples: Fibonacci $\text{fib}(n) = \text{fib}(n - 1) + \text{fib}(n - 2)$ with $\text{fib}(0) = 0$, $\text{fib}(1) = 1$; GCD $\text{gcd}(a, b) = \text{gcd}(b, a \% b)$ with $\text{gcd}(a, 0) = a$; sum of digits; power $x^n = x \times x^{n-1}$; Tower of Hanoi ($2^n - 1$ moves).

Recursion	Iteration
Function calls itself	Uses loops
Needs a base case	Needs a loop condition
Uses the stack; more memory and function-call overhead	Less memory, usually faster
Short, elegant code for problems like trees, Hanoi, quick sort	Better for simple repetition
Missing base case causes stack overflow	Missing condition causes an infinite loop

2.3 Arrays

An **array** is a collection of elements of the same data type stored in **contiguous memory locations** and accessed by a common name with an index (subscript).

One-dimensional array.

- Declaration: `int marks[5];` (5 integers, indices 0 to 4).
- Initialisation: `int a[5] = {10, 20, 30, 40, 50};` `int b[] = {1, 2, 3};` (size 3); `int c[5] = {1, 2};` (rest are 0).
- Access: `a[0]` is the first element, `a[4]` the last. C does **no bounds checking**: accessing `a[5]` is an error that the compiler does not catch.
- Memory: address of `a[i] = \text{base address} + i \times \text{sizeof}(\text{element})`. For `int a[5]` at address 1000 (4-byte int), `a[3]` is at 1012.

Two-dimensional array. `int m[3][4];` (3 rows, 4 columns). Stored in **row-major order** (row 0 first). Initialisation: `int m[2][3] = {{1, 2, 3}, {4, 5, 6}};` Address of `m[i][j] = \text{base} + (i \times \text{columns} + j) \times \text{size}`.

Common array operations: traversal, insertion, deletion, searching, sorting, finding max/min/sum/average, reversing, merging.

2.4 Matrix operations

- **Addition:** `C[i][j] = A[i][j] + B[i][j]` (same order required).
- **Transpose:** `T[j][i] = A[i][j]`.

- **Multiplication:** $A (m \times n) \times B (n \times p) = C (m \times p)$, with $C[i][j] = \sum_k A[i][k] \times B[k][j]$. Possible only if the number of columns of A equals the number of rows of B. It needs three nested loops.

```
#include <stdio.h>
int main(void)
{
    int a[2][3] = {{1, 2, 3}, {4, 5, 6}};
    int b[3][2] = {{7, 8}, {9, 10}, {11, 12}};
    int c[2][2] = {{0}};
    int i, j, k;
    for (i = 0; i < 2; i++)
        for (j = 0; j < 2; j++)
            for (k = 0; k < 3; k++)
                c[i][j] += a[i][k] * b[k][j];
    for (i = 0; i < 2; i++) {
        for (j = 0; j < 2; j++) printf("%5d", c[i][j]);
        printf("\n");
    }
    return 0;
}
```

Output: 58 64 / 139 154.

2.5 Strings

A **string** is a one-dimensional array of characters terminated by the null character '\0'.

- Declaration: `char name[20]`; Initialisation: `char s[] = "INDIA"`; (6 bytes) or `char s[] = {'I', 'N', 'D', 'I', 'A', '\0'}`;
- Input: `scanf("%s", name)`; (stops at a space; no &) or `fgets(name, 20, stdin)`; (reads a whole line including spaces).
- Output: `printf("%s", name)`; or `puts(name)`;

String library functions (string.h).

Function	Purpose	Example
<code>strlen(s)</code>	Length (excluding '\0')	<code>strlen("hello") = 5</code>
<code>strcpy(d, s)</code>	Copy s into d	<code>strcpy(d, "abc")</code>
<code>strcat(d, s)</code>	Append s to the end of d	"Hello" + "World" → "HelloWorld"
<code>strcmp(s1, s2)</code>	Compare: 0 if equal, <0 if s1 < s2, >0 if s1 > s2	<code>strcmp("abc", "abd") < 0</code>
<code>strrev(s)</code>	Reverse (non-standard; not in gcc)	write your own
<code>strupr / strlwr</code>	Upper / lower case (non-standard)	use <code>toupper()/tolower()</code> in a loop
<code>strchr(s, c) / strstr(s1, s2)</code>	Find a character / substring	return a pointer or NULL

Example: palindrome check without library functions.

```
#include <stdio.h>
int main(void)
{
    char s[100];
    int i = 0, j = 0, pal = 1;
    printf("Enter a word: ");
    scanf("%99s", s);
    while (s[j] != '\0') j++;          /* length */
    for (j = j - 1; i < j; i++, j--)
        if (s[i] != s[j]) { pal = 0; break; }
    printf("%s is %sa palindrome\n", s, pal ? "" : "not ");
    return 0;
}
```

Input MADAM → MADAM is a palindrome.

2.6 Passing arrays to functions

When an array is passed to a function, only the **address of its first element** is passed (not a copy of the whole array). So changes made to array elements inside the function **do** affect the original array.

```
#include <stdio.h>
void doubleAll(int a[], int n)      /* int a[] is the same as int *a */
{
    int i;
    for (i = 0; i < n; i++) a[i] *= 2;
}
int maxOf(const int a[], int n)     /* const: function will not modify */
{
    int i, m = a[0];
    for (i = 1; i < n; i++) if (a[i] > m) m = a[i];
    return m;
}
int main(void)
{
    int x[5] = {3, 9, 1, 7, 5}, i;
    doubleAll(x, 5);
    for (i = 0; i < 5; i++) printf("%d ", x[i]);
    printf("\nMax = %d\n", maxOf(x, 5));
    return 0;
}
```

Output: 6 18 2 14 10 / Max = 18. The size must be passed separately, because inside the function `sizeof(a)` gives the size of a pointer, not of the array.

2-D arrays: the column size must be specified: `void print(int m[][3], int rows);`

2.7 Structures

A **structure** is a user-defined data type that groups variables of **different types** under one name. Example: a student record with roll number (int), name (string) and marks (float).

```
struct student {                  /* structure template (no memory yet) */
    int roll;
    char name[30];
    float marks;
};
struct student s1 = {101, "Asha", 88.5}; /* variable with initialisation */
```



- **Member access:** dot operator: s1.roll, s1.name, s1.marks. For a pointer to a structure, use the arrow operator: p->marks (same as (*p).marks).
- **Size:** at least the sum of member sizes; the compiler may add padding for alignment. Use sizeof(struct student).
- **Array of structures:** struct student cls[60]; cls[i].marks.
- **Nested structures:** a structure member can itself be a structure (e.g. struct date dob; inside struct employee).
- **Structures and functions:** a structure can be passed by value (copy) or by address (pointer); a function can return a structure.
- **typedef:** typedef struct student Student; then Student s2; (shorter name).
- Structures can be assigned (s2 = s1;) but cannot be compared with == directly.

```
#include <stdio.h>
struct student { int roll; char name[30]; float marks; };
int main(void)
{
    struct student s[3];
    int i, top = 0;
    for (i = 0; i < 3; i++) {
        printf("Roll, name, marks: ");
        scanf("%d %29s %f", &s[i].roll, s[i].name, &s[i].marks);
    }
    for (i = 1; i < 3; i++) if (s[i].marks > s[top].marks) top = i;
    printf("Topper: %s (roll %d) with %.1f marks\n", s[top].name, s[top].roll, s[top].marks);
    return 0;
}
```

2.8 Unions

A **union** is like a structure, but all its members **share the same memory location**. Its size equals the size of its largest member. Only one member holds a meaningful value at a time.

```
#include <stdio.h>
union data { int i; float f; char c; };
struct sdata { int i; float f; char c; };
int main(void)
{
    union data u;
    printf("sizeof union = %zu, sizeof struct = %zu\n", sizeof(union data), sizeof(struct sdata));
    u.i = 65;
    printf("u.i = %d, u.c = %c\n", u.i, u.c); /* c shares the low byte of i */
    u.f = 2.5f; /* now i is overwritten */
    printf("u.f = %.1f\n", u.f);
    return 0;
}
```

Typical output (little-endian PC): sizeof union = 4, sizeof struct = 12; u.i = 65, u.c = A; u.f = 2.5.

Structure	Union
Keyword struct	Keyword union
Each member has its own memory	All members share the same memory
Size $\geq$ sum of member sizes	Size = size of the largest member



Structure	Union
All members can hold values simultaneously	Only one member is valid at a time
Used to group related data (records)	Used to save memory, or to view the same data in different ways (e.g. device registers, variant records)

Enumeration (enum): enum color {RED, GREEN, BLUE}; gives named integer constants (RED = 0, GREEN = 1, BLUE = 2), improving readability.

2.9 Common mistakes to avoid in Module II

- Forgetting the base case in recursion (stack overflow).
- Accessing a[n] for an array of size n (valid indices are 0 to n – 1).
- Using = instead of strcpy() to copy strings (s1 = s2; is illegal for arrays).
- Comparing strings with == (compares addresses). Use strcmp().
- Forgetting the extra byte for '\0' when declaring a char array.
- Using & with a string in scanf("%s", &name) (works by accident, but name alone is correct).

2.10 Solved problems: Module II

Problem 1. Write a program to print the Fibonacci series using a recursive function.

```
#include <stdio.h>
int fib(int n)
{
    if (n == 0) return 0;
    if (n == 1) return 1;
    return fib(n - 1) + fib(n - 2);
}
int main(void)
{
    int n, i;
    printf("How many terms? ");
    scanf("%d", &n);
    for (i = 0; i < n; i++) printf("%d ", fib(i));
    printf("\n");
    return 0;
}
```

ANS: For n = 8: 0 1 1 2 3 5 8 13. (Recursive Fibonacci is slow for large n because it recomputes values; the loop version is faster.)

Problem 2. Write a recursive function to find the GCD of two numbers.

```
#include <stdio.h>
int gcd(int a, int b)
{
    return b == 0 ? a : gcd(b, a % b);
}
int main(void)
{
    printf("GCD(48, 18) = %d\n", gcd(48, 18));
    return 0;
}
```

S: $\text{gcd}(48, 18) \rightarrow \text{gcd}(18, 12) \rightarrow \text{gcd}(12, 6) \rightarrow \text{gcd}(6, 0) = 6$.

ANS: $\text{GCD}(48, 18) = 6$.

Problem 3. Write a program to swap two numbers using a function. Explain why call by value does not work.

```
#include <stdio.h>
void swapByValue(int a, int b) { int t = a; a = b; b = t; }
void swapByRef(int *a, int *b) { int t = *a; *a = *b; *b = t; }
int main(void)
{
    int x = 10, y = 20;
    swapByValue(x, y);
    printf("After swapByValue: x = %d, y = %d\n", x, y);
    swapByRef(&x, &y);
    printf("After swapByRef: x = %d, y = %d\n", x, y);
    return 0;
}
```

S: swapByValue exchanges only its local copies; x and y in main are unchanged. swapByRef receives the addresses and exchanges the actual variables.

ANS: After swapByValue: x = 10, y = 20; after swapByRef: x = 20, y = 10.

Problem 4. Write a program to find the largest, smallest, sum and average of n numbers in an array.

```
#include <stdio.h>
int main(void)
{
    int a[100], n, i, max, min;
    long sum = 0;
    printf("Enter n: ");
    scanf("%d", &n);
    for (i = 0; i < n; i++) scanf("%d", &a[i]);
    max = min = a[0];
    for (i = 0; i < n; i++) {
        if (a[i] > max) max = a[i];
        if (a[i] < min) min = a[i];
        sum += a[i];
    }
    printf("Max = %d, Min = %d, Sum = %ld, Average = %.2f\n", max, min, sum, (float)sum / n);
    return 0;
}
```

ANS: For 5 numbers 4 9 1 7 3: Max = 9, Min = 1, Sum = 24, Average = 4.80.

Problem 5. Write a program to insert an element at a given position in an array.

```
#include <stdio.h>
int main(void)
{
    int a[50] = {10, 20, 30, 40, 50}, n = 5, pos, val, i;
    printf("Enter position (1-%d) and value: ", n + 1);
    scanf("%d %d", &pos, &val);
    for (i = n; i >= pos; i--) a[i] = a[i - 1];    /* shift right */
    a[pos - 1] = val;
    n++;
    for (i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
    return 0;
}
```

ANS: Position 3, value 25 → 10 20 25 30 40 50. (Deletion is the reverse: shift elements left from the position.)

Problem 6. Write a program to find the transpose of a matrix and check whether it is symmetric.

```
#include <stdio.h>
int main(void)
{
    int a[3][3] = {{1, 2, 3}, {2, 5, 6}, {3, 6, 9}}, t[3][3], i, j, sym = 1;
    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++) {
            t[j][i] = a[i][j];
        }
    for (i = 0; i < 3; i++)
        for (j = 0; j < 3; j++)
            if (a[i][j] != t[i][j]) sym = 0;
    printf("The matrix is %ssymmetric\n", sym ? "" : "not ");
    return 0;
}
```

ANS: The matrix is symmetric (compare Mathematics-I, Module V).

Problem 7. Write a program to count vowels, consonants, digits and spaces in a line of text.

```
#include <stdio.h>
#include <ctype.h>
int main(void)
{
    char s[200];
    int i, v = 0, c = 0, d = 0, sp = 0;
    printf("Enter a line: ");
    fgets(s, sizeof s, stdin);
    for (i = 0; s[i] != '\0'; i++) {
        char ch = tolower((unsigned char)s[i]);
        if (ch == 'a' || ch == 'e' || ch == 'i' || ch == 'o' || ch == 'u') v++;
        else if (isalpha((unsigned char)ch)) c++;
        else if (isdigit((unsigned char)ch)) d++;
        else if (ch == ' ') sp++;
    }
    printf("Vowels=%d Consonants=%d Digits=%d Spaces=%d\n", v, c, d, sp);
    return 0;
}
```

ANS: "GSE Brahmapur 2023" → Vowels = 4, Consonants = 8, Digits = 4, Spaces = 2.

Problem 8. Write your own versions of strlen, strcpy and strcat.

```
#include <stdio.h>
int myStrlen(const char s[]) { int n = 0; while (s[n]) n++; return n; }
void myStrcpy(char d[], const char s[]) { int i = 0; while ((d[i] = s[i]) != '\0') i++; }
void myStrcat(char d[], const char s[]) { int i = myStrlen(d), j = 0; while ((d[i++] = s[j++]) != '\0') j++; }
int main(void)
{
    char a[50], b[] = "World";
    myStrcpy(a, "Hello ");
    myStrcat(a, b);
    printf("%s (length %d)\n", a, myStrlen(a));
    return 0;
}
```

ANS: Hello World (length 11).

Problem 9. Define a structure for an employee (id, name, basic pay) and compute gross salary = basic + DA (50%) + HRA (20%) for n employees.

```
#include <stdio.h>
struct emp { int id; char name[30]; float basic, gross; };
int main(void)
{
    struct emp e[10];
    int n, i;
    printf("Number of employees: ");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        scanf("%d %29s %f", &e[i].id, e[i].name, &e[i].basic);
        e[i].gross = e[i].basic * (1 + 0.50f + 0.20f);
    }
    for (i = 0; i < n; i++)
        printf("%d %-10s %10.2f\n", e[i].id, e[i].name, e[i].gross);
    return 0;
}
```

ANS: For basic 20000, gross = 34000.00.



Problem 10. Predict the output: `int f(int n) { if (n == 0) return 0; return n % 10 + f(n / 10); } ... printf("%d", f(4567));`

$f(4567) = 7 + f(456) = 7 + 6 + f(45) = 7 + 6 + 5 + 4 + f(0) = 22$.

Answer: 22 (sum of digits).

Problem 11. Predict the output: `int a[5] = {1, 2, 3}; printf("%d %d", a[1], a[4]);`

Unspecified elements of a partially initialised array are 0.

Answer: 2 0

Problem 12. Predict the output: `char s[] = "C Prog"; printf("%zu %zu", sizeof(s), strlen(s));`

`sizeof` includes `'\0'` (7 bytes); `strlen` does not (6).

Answer: 7 6

Problem 13. Predict the output: `void f(int x) { x = 100; } int main() { int a = 5; f(a); printf("%d", a); }`

Call by value: `f` changes only its copy.

Answer: 5

2.11 Practice problems with solutions: Module II

Q1. Write a recursive function to compute x^n for integer $n \geq 0$.

`long power(int x, int n) { if (n == 0) return 1; return x * power(x, n - 1); }`

Answer: `power(2, 10) = 1024`.

Q2. Write a recursive function to reverse a number (print digits in reverse).

`void rev(int n) { if (n == 0) return; printf("%d", n % 10); rev(n / 10); }`

Answer: `rev(1234)` prints 4321.

Q3. How many moves does the Tower of Hanoi need for 4 discs? Write the recursive function.

Moves = $2^n - 1 = 15$. `void hanoi(int n, char from, char to, char via) { if (n == 0) return; hanoi(n - 1, from, via, to); printf("Move disc %d from %c to %c\n", n, from, to); hanoi(n - 1, via, to, from); }`

Answer: 15 moves.

Q4. Find the address of `a[2][3]` in `int a[4][5]` stored row-major at base address 2000 (4-byte int).

Address = $2000 + (2 \times 5 + 3) \times 4 = 2000 + 52$.

Answer: 2052.

Q5. Write a program segment to reverse an array in place.

```
for (i = 0, j = n - 1; i < j; i++, j--) { t = a[i]; a[i] = a[j]; a[j] = t; }
```

Answer: {1, 2, 3, 4, 5} becomes {5, 4, 3, 2, 1}.

Q6. Write a program segment to find the sum of the diagonal elements (trace) of an $n \times n$ matrix.

```
trace = 0; for (i = 0; i < n; i++) trace += a[i][i];
```

Answer: For {{1, 2}, {3, 4}}, trace = 5.

Q7. What is the output of: `printf("%d", strcmp("apple", "apricot") < 0);`

Comparison stops at the third character: 'p' (112) vs 'r' (114); 'p' < 'r', so the result is negative.

Answer: 1

Q8. What is the size of: `struct s { char c; int i; };` on a typical system? Why?

1 byte for c + 3 bytes padding (to align i on a 4-byte boundary) + 4 bytes for i.

Answer: 8 bytes (not 5).

Q9. What is the size of union `u { char c[10]; int i; double d; };` ?

Largest member is char c[10] (10 bytes); the size is rounded up to a multiple of the largest alignment (8 for double).

Answer: 16 bytes on a typical 64-bit system.

Q10. Write a function that counts the words in a string.

```
int words(const char s[]) { int i, n = 0, in = 0; for (i = 0; s[i]; i++) { if (s[i] == ' ' || s[i] == '\n' || s[i] == '\t') in = 0; else if (!in) { in = 1; n++; } } return n; }
```

Answer: `words("I love C programming") = 4.`

2.12 MCQs: Module II

1. A function that calls itself is called

- (a) inline (b) recursive (c) iterative (d) static

☐ (b) — Definition.

2. A recursive function must have

- (a) a loop (b) a pointer (c) a base case (d) a global variable

☐ (c) — To terminate.

3. Missing base case in recursion leads to

- (a) zero output (b) faster execution (c) stack overflow at run time (d) compile error

☐ (c) — Infinite calls.

4. The default parameter passing mechanism in C is

- (a) call by name (b) call by reference (c) call by result (d) call by value

☐ (d) — Copies are passed.

5. Values passed to a function in the call are

- (a) global variables (b) formal parameters (c) local variables (d) actual arguments

☐ (d) — In the caller.



- 6.** The return type of a function that returns nothing is
(a) null (b) empty (c) int (d) void
☐ (d) — No value.
- 7.** A function prototype tells the compiler
(a) the memory address (b) the body of the function (c) the output (d) the name, return type and parameter types
☐ (d) — Declaration.
- 8.** Array indices in C start from
(a) any value (b) 1 (c) -1 (d) 0
☐ (d) — Zero-based.
- 9.** For `int a[10];`, the last valid element is
(a) `a[0]` (b) `a[10]` (c) `a[11]` (d) `a[9]`
☐ (d) — Indices 0-9.
- 10.** Arrays in C are stored in
(a) random locations (b) registers (c) contiguous memory (d) linked nodes
☐ (c) — Consecutive addresses.
- 11.** 2-D arrays in C are stored in
(a) column-major order (b) random order (c) row-major order (d) diagonal order
☐ (c) — Rows one after another.
- 12.** `int a[5] = {1, 2};` The value of `a[3]` is
(a) garbage (b) error (c) 2 (d) 0
☐ (d) — Remaining elements are zero.
- 13.** When an array is passed to a function, what is passed?
(a) the address of the first element (b) a copy of all elements (c) only the first element (d) the size
☐ (a) — Array decays to pointer.
- 14.** Matrix multiplication of A (2×3) and B (3×4) gives a matrix of order
(a) 3×3 (b) 4×2 (c) 2×3 (d) 2×4
☐ (d) — $m \times p$.
- 15.** A string in C is terminated by
(a) `'\n'` (b) `'\0'` (c) EOF (d) `'.'`
☐ (b) — Null character.
- 16.** `strlen("Engineering")` returns
(a) 12 (b) 11 (c) 10 (d) 13
☐ (b) — Excludes `'\0'`.
- 17.** `strcmp("abc", "abc")` returns
(a) 1 (b) 0 (c) 3 (d) -1
☐ (b) — Equal strings.
- 18.** The function to join two strings is
(a) `strjoin` (b) `strcmp` (c) `strcat` (d) `strcpy`
☐ (c) — Concatenate.
- 19.** `scanf("%s", str)` stops reading at
(a) 10 characters (b) newline only (c) a white space (d) `'\0'`
☐ (c) — Use `fgets` for lines.
- 20.** Which header file contains `strlen()`?
(a) `string.h` (b) `math.h` (c) `stdlib.h` (d) `stdio.h`
☐ (a) — String functions.



- 21.** Which is used to access a structure member through a pointer?
(a) * (b) & (c) -> (d) .
☐ (c) — Arrow operator.
- 22.** The size of a union is
(a) sum of member sizes (b) size of the smallest member (c) size of the largest member (with padding) (d) always 4
☐ (c) — Members share memory.
- 23.** In a union, how many members can hold a valid value at a time?
(a) all (b) two (c) one (d) none
☐ (c) — Shared storage.
- 24.** The keyword used to create an alias name for a data type is
(a) define (b) alias (c) struct (d) typedef
☐ (d) — typedef.
- 25.** Structures can contain members of
(a) only the same type (b) only arrays (c) different types (d) only int
☐ (c) — Heterogeneous.
- 26.** Which operation is not allowed directly on structures?
(a) returning from a function (b) passing to a function (c) assignment (d) comparison with ==
☐ (d) — Compare member by member.
- 27.** enum colour {RED, GREEN, BLUE}; The value of BLUE is
(a) 1 (b) 2 (c) 0 (d) 3
☐ (b) — Starts from 0.
- 28.** The number of moves for Tower of Hanoi with n discs is
(a) n! (b) $2n - 1$ (c) $2^n - 1$ (d) n^2
☐ (c) — Recurrence.
- 29.** sqrt() is declared in
(a) stdio.h (b) math.h (c) stdlib.h (d) string.h
☐ (b) — Link with -lm.
- 30.** A local variable declared inside a function is visible
(a) in all files (b) in main only (c) everywhere (d) only inside that function (block)
☐ (d) — Block scope.
- 31.** Recursion is generally
(a) faster than iteration (b) impossible in C (c) slower and uses more memory than iteration (d) the same as iteration
☐ (c) — Function-call overhead.
- 32.** An array of structures is useful to store
(a) different programs (b) many records of the same type (c) one record (d) functions
☐ (b) — E.g. class list.
- 33.** The output of `int a[] = {5, 10, 15}; printf("%d", a[1] + a[2]);` is
(a) 15 (b) 30 (c) 25 (d) 20
☐ (c) — 10 + 15.
- 34.** Which declaration of a string is correct?
(a) `char *s[] = 'hi';` (b) `char s[] = "hi";` (c) `string s = "hi";` (d) `char s = "hi";`
☐ (b) — Character array.



- 35.** sizeof(char[20]) is
(a) 4 (b) 19 (c) 20 (d) 21
☐ (c) — Array of 20 chars.
- 36.** A function can return
(a) multiple values directly (b) one value (or a structure) (c) only int (d) nothing ever
☐ (b) — Use pointers/structures for more.
- 37.** Passing a large structure by pointer rather than by value
(a) wastes memory (b) saves time and memory (c) is illegal (d) changes nothing
☐ (b) — No copying.
- 38.** The main() function returns 0 to indicate
(a) infinite loop (b) error (c) nothing (d) successful termination
☐ (d) — Convention.
- 39.** toupper('a') returns
(a) 'A' (b) 65.0 (c) error (d) 'a'
☐ (a) — ctype.h.
- 40.** A nested structure means
(a) a union inside a loop (b) a structure inside a function (c) two structures with the same name (d) a structure member that is itself a structure
☐ (d) — E.g. date of birth.

Module III: Pointers and Dynamic Memory Allocation

3.1 Pointer basics

Every variable occupies memory at some **address**. A **pointer** is a variable that stores the address of another variable.

```
int x = 25;
int *p;      /* p is a pointer to int */
p = &x;      /* &x gives the address of x; p now points to x */
```

- **& (address-of operator):** &x is the address of x.
- **Indirection (dereference) operator *:** *p is the value stored at the address in p, i.e. the value of x. Changing *p changes x.
- **Printing:** printf("%p", (void *)p); prints an address; printf("%d", *p); prints 25.

Why pointers? Call by reference (functions that modify the caller's variables or return several values); efficient handling of arrays and strings; dynamic memory allocation; building data structures such as linked lists and trees; accessing hardware addresses in embedded systems.

Pointer declaration and types. The pointer's type tells the compiler how many bytes to read at that address and how pointer arithmetic works: int *ip; float *fp; char *cp; All pointers have the same size (4 bytes on 32-bit, 8 bytes on 64-bit systems), whatever they point to.

- **NULL pointer:** int *p = NULL; points to nothing. Always check for NULL before dereferencing.
- **void pointer (generic pointer):** void *vp; can hold the address of any type, but must be cast before dereferencing. malloc() returns void *.
- **Wild pointer:** an uninitialised pointer (contains a garbage address). Dereferencing it is dangerous.
- **Dangling pointer:** a pointer to memory that has been freed or to a local variable that no longer exists.

```
#include <stdio.h>
int main(void)
{
    int x = 25, *p = &x;
    printf("x = %d, *p = %d\n", x, *p);
    *p = 40;                      /* changes x through p */
    printf("After *p = 40, x = %d\n", x);
    printf("sizeof(p) = %zu bytes\n", sizeof(p));
    return 0;
}
```

Output: x = 25, *p = 25 / After *p = 40, x = 40 / sizeof(p) = 8 bytes (on a 64-bit system).

3.2 Pointer arithmetic

Only a few operations are allowed on pointers:

- **Adding or subtracting an integer:** p + 1 points to the **next element**, i.e. the address increases by sizeof(type). For int *p at 1000 (4-byte int), p + 1 = 1004, p + 3 = 1012. For char *, p + 1 = 1001; for double *, p + 1 = 1008.
- **Increment/decrement:** p++, p--.

- **Subtracting two pointers** (of the same array): gives the number of elements between them.
- **Comparing pointers** (of the same array) with <, >, ==, !=.

Not allowed: adding two pointers, multiplying or dividing pointers, adding a float to a pointer.

Precedence traps: *p++ means *(p++) (use the value, then move p); (*p)++ increments the value pointed to; ++*p increments the value; *++p moves p first, then reads.

3.3 Pointers and arrays

The name of an array is a **constant pointer** to its first element: a is the same as &a[0].

$a[i] \equiv *(a + i)$ & $\&a[i] \equiv a + i$ (and even $i[a]$ works, since $a[i] = *(a + i) = *(i + a)$)

```
#include <stdio.h>
int main(void)
{
    int a[5] = {10, 20, 30, 40, 50}, *p = a, i;
    for (i = 0; i < 5; i++)
        printf("%d %d %d\n", a[i], *(a + i), *(p + i));
    p = &a[2];
    printf("*p = %d, *(p+1) = %d, p - a = %td\n", *p, *(p + 1), p - a);
    return 0;
}
```

Differences between an array name and a pointer: a pointer can be changed (p++ is legal), but an array name cannot (a++ is illegal); sizeof(a) gives the whole array size, sizeof(p) the pointer size.

Pointers and strings. char *s = "Hello"; points to a string literal (read-only; do not modify). char t[] = "Hello"; is a modifiable array. A string can be traversed with a pointer: while (*s) putchar(*s++);

Array of pointers: char *days[] = {"Mon", "Tue", "Wed"}; stores addresses of strings (saves memory compared with a 2-D char array).

Pointer to an array: int (*pa)[5]; points to a whole array of 5 ints (used with 2-D arrays).

3.4 Parameter passing using pointers: call by value vs call by reference

Call by value	Call by reference (using pointers)
Copies of the values of actual arguments are passed	Addresses of actual arguments are passed
Changes inside the function do not affect the caller's variables	Changes made through the pointers affect the caller's variables
Formal parameters are ordinary variables: void f(int x)	Formal parameters are pointers: void f(int *x)
Call: f(a)	Call: f(&a)
Safer (original data protected)	Can return multiple results; efficient for large data

Example: function returning several results (sum and product) through pointers.

```
#include <stdio.h>
void sumProd(int a, int b, int *s, int *p)
{
    *s = a + b;
    *p = a * b;
}
int main(void)
{
    int s, p;
    sumProd(6, 7, &s, &p);
    printf("Sum = %d, Product = %d\n", s, p);
    return 0;
}
```

Output: Sum = 13, Product = 42.

3.5 Pointer to pointer

A pointer to pointer stores the address of another pointer: `int x = 5, *p = &x, **pp = &p;` Then `*pp = p, **pp = x = 5`. Used for dynamic 2-D arrays, arrays of strings (`char **argv`), and functions that must change a pointer (e.g. inserting at the head of a linked list).

```
#include <stdio.h>
int main(void)
{
    int x = 5, *p = &x, **pp = &p;
    **pp = 50;
    printf("x = %d, *p = %d, **pp = %d\n", x, *p, **pp);
    return 0;
}
```

Output: x = 50, *p = 50, **pp = 50.

3.6 Pointers to structures and functions

Pointer to structure. `struct student s, *ps = &s;` members are accessed with `ps->marks` (equivalent to `(*ps).marks`; the parentheses are needed because `.` has higher precedence than `*`).

Pointer to function. `int (*fp)(int, int) = add;` then `fp(2, 3)` calls `add`. Used for callbacks (e.g. the comparison function passed to the library `qsort`) and menu-driven programs with a table of functions.

3.7 Static vs dynamic memory

Static (compile-time) allocation: the size of variables and arrays is fixed when the program is compiled (`int a[100];`). Too little memory causes overflow; too much wastes memory. Local variables are stored on the **stack** and are freed automatically when the function returns.

Dynamic (run-time) allocation: memory is requested while the program runs, in exactly the amount needed, from a region called the **heap**. It remains allocated until the program frees it with `free()`. This is essential for data structures whose size changes at run time (linked lists, trees) and for arrays whose size is known only at run time.



Static memory	Dynamic memory
Allocated at compile time	Allocated at run time
Size fixed	Size can be decided and changed at run time
Stack (locals) or data segment (globals/static)	Heap
Freed automatically	Must be freed by the programmer with free()
Accessed by variable names	Accessed only through pointers
Faster, simpler	Flexible; risk of memory leaks

3.8 Dynamic memory allocation functions (stdlib.h)

Function	Syntax	Description
malloc()	ptr = (int *) malloc(n * sizeof(int));	Allocates a single block of n × size bytes; contents are uninitialised (garbage); returns NULL if it fails
calloc()	ptr = (int *) calloc(n, sizeof(int));	Allocates n blocks, each of the given size, and initialises all bytes to zero
realloc()	ptr = (int *) realloc(ptr, newsize);	Changes the size of a previously allocated block (grows or shrinks); may move it and copy the old contents
free()	free(ptr);	Releases the memory back to the heap; afterwards set ptr = NULL

malloc vs calloc.

malloc	calloc
One argument (total bytes)	Two arguments (number of elements, size of each)
Memory is not initialised (garbage values)	Memory is initialised to zero
Slightly faster	Slightly slower (zero-filling)

Memory leak: memory that is allocated but never freed, and whose address is lost; the program's memory use keeps growing. **Rule:** every malloc/calloc must have a matching free.

Program: dynamic array of n integers (malloc, realloc, free).

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int n, i, *a;
    printf("How many numbers? ");
    if (scanf("%d", &n) != 1 || n <= 0) return 1;
    a = (int *) malloc(n * sizeof(int));
    if (a == NULL) { printf("Memory not available\n"); return 1; }
    for (i = 0; i < n; i++) a[i] = (i + 1) * 10;      /* fill: 10, 20, ... */
    a = (int *) realloc(a, (n + 2) * sizeof(int)); /* grow by 2 */
    if (a == NULL) return 1;
    a[n] = 111; a[n + 1] = 222;
    for (i = 0; i < n + 2; i++) printf("%d ", a[i]);
    printf("\n");
    free(a);
    a = NULL;
    return 0;
}
```

Input 3 → Output: 10 20 30 111 222. (In real programs, store realloc's result in a temporary pointer first, so the original block is not lost if realloc fails.)

Program: dynamic 2-D array (matrix of $r \times c$).

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int r = 2, c = 3, i, j;
    int **m = malloc(r * sizeof(int *));
    for (i = 0; i < r; i++) m[i] = calloc(c, sizeof(int));
    for (i = 0; i < r; i++)
        for (j = 0; j < c; j++) m[i][j] = i * c + j;
    for (i = 0; i < r; i++) {
        for (j = 0; j < c; j++) printf("%d ", m[i][j]);
        printf("\n");
        free(m[i]);
    }
    free(m);
    return 0;
}
```

Output: 0 1 2 / 3 4 5.

3.9 Common mistakes with pointers

- Using an uninitialised (wild) pointer: `int *p; *p = 5;` (undefined behaviour).
- Forgetting `&` in `scanf` for non-array variables.
- Returning the address of a local variable from a function (it becomes dangling).
- Forgetting to check whether `malloc` returned `NULL`.
- Forgetting `free()` (memory leak) or freeing the same block twice.
- Using memory after `free()` (dangling pointer).
- Writing `*p++` when `(*p)++` is intended.

3.10 Solved problems: Module III

Problem 1. Write a program to read an array of n numbers and find the smallest number and its position, using a pointer and a function.

```
#include <stdio.h>
void smallest(const int *a, int n, int *min, int *pos)
{
    int i;
    *min = *a; *pos = 0;
    for (i = 1; i < n; i++)
        if (*(a + i) < *min) { *min = *(a + i); *pos = i; }
}
int main(void)
{
    int a[50], n, i, min, pos;
    printf("Enter n and the numbers: ");
    scanf("%d", &n);
    for (i = 0; i < n; i++) scanf("%d", a + i);
    smallest(a, n, &min, &pos);
    printf("Smallest = %d at position %d\n", min, pos + 1);
    return 0;
}
```

ANS: For 5 numbers 8 3 9 1 6: Smallest = 1 at position 4.

Problem 2. Write a program to check whether a string is a palindrome using pointers.

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char s[100], *l, *r;
    printf("Enter a string: ");
    scanf("%99s", s);
    l = s; r = s + strlen(s) - 1;
    while (l < r && *l == *r) { l++; r--; }
    printf("%s\n", l >= r ? "Palindrome" : "Not a palindrome");
    return 0;
}
```

ANS: "racecar" → Palindrome; "radar1" → Not a palindrome.

Problem 3. Write a program to find the sum of array elements using pointer arithmetic only (no []).

```
#include <stdio.h>
int main(void)
{
    int a[6] = {4, 8, 15, 16, 23, 42}, *p, sum = 0;
    for (p = a; p < a + 6; p++) sum += *p;
    printf("Sum = %d\n", sum);
    return 0;
}
```

ANS: Sum = 108.

Problem 4. Write a program to allocate memory for n integers using calloc, read them, and print them in reverse, then free the memory.

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int n, i, *p;
    printf("Enter n: ");
    scanf("%d", &n);
    p = (int *) calloc(n, sizeof(int));
    if (!p) { printf("Allocation failed\n"); return 1; }
    for (i = 0; i < n; i++) scanf("%d", p + i);
    for (i = n - 1; i >= 0; i--) printf("%d ", p[i]);
    printf("\n");
    free(p);
    return 0;
}
```

ANS: Input 4 numbers 1 2 3 4 → 4 3 2 1.

Problem 5. Write a program to store student records dynamically using a pointer to structure.

```
#include <stdio.h>
#include <stdlib.h>
struct stud { int roll; float cgpa; };
int main(void)
{
    int n, i;
    struct stud *s;
    printf("Number of students: ");
    scanf("%d", &n);
    s = malloc(n * sizeof *s);
    if (!s) return 1;
    for (i = 0; i < n; i++) scanf("%d %f", &(s + i)->roll, &(s + i)->cgpa);
    for (i = 0; i < n; i++) printf("Roll %d: CGPA %.2f\n", (s + i)->roll, s[i].cgpa);
    free(s);
    return 0;
}
```

Problem 6. Predict the output: int a[] = {10, 20, 30, 40}; int *p = a; printf("%d", *p++); printf("%d ", *p); printf("%d", ++*p);

*p++ prints 10 and moves p to a[1]; *p is 20; ++*p increments a[1] to 21 and prints it.

Answer: 10 20 21

Problem 7. Predict the output: int x = 10, *p = &x, **q = &p; **q = **q + 5; printf("%d %d", x, *p);

**q refers to x. x becomes 15.

Answer: 15 15

Problem 8. Predict the output: `char *s = "COMPUTER"; printf("%c %c %s", *s, *(s + 3), s + 5);`

`*s = 'C'; s[3] = 'P'; s + 5` points to "TER".

Answer: C P TER

Problem 9. Predict the output (64-bit gcc): `double d[5]; double *p = d; printf("%ld", (long)((char *) (p + 2) - (char *) p));`

`p + 2` is 2 doubles ahead = 16 bytes.

Answer: 16

Problem 10. Predict the output: `int a[3][3] = {{1,2,3},{4,5,6},{7,8,9}}; printf("%d %d", *(a + 1) + 2, **a);`

`*(a + 1)` is row 1; `+ 2` → element `a[1][2] = 6`. `**a = a[0][0] = 1`.

Answer: 6 1

Problem 11. What is wrong with this function? `int *f(void) { int x = 5; return &x; }`

`x` is a local (automatic) variable; it is destroyed when `f` returns, so the returned pointer is dangling.

Answer: Returns the address of a local variable (undefined behaviour). Use `static int x`, or allocate with `malloc`.

Problem 12. What is wrong? `int *p = malloc(10 * sizeof(int)); p = malloc(20 * sizeof(int));`

The address of the first block is lost without `free()`.

Answer: Memory leak of 40 bytes (on a 4-byte-int system). Free the first block or use `realloc`.

3.11 Practice problems with solutions: Module III

Q1. If `int a[5]` starts at address 2000 (4-byte int) and `int *p = a + 1`, what are `p`, `p + 2` and `*(p + 2)` if `a = {5, 10, 15, 20, 25}`?

`p = 2004; p + 2 = 2012` (element `a[3]`); `*(p + 2) = 20`.

Answer: 2004, 2012, 20.

Q2. Write a function using pointers that converts a string to upper case.

```
void upper(char *s) { while (*s) { if (*s >= 'a' && *s <= 'z') *s -= 32; s++; } }
```

Answer: "Gse brahmapur" becomes "GSE BRAHMAPUR".

Q3. Write a function to count the characters in a string using pointer arithmetic.

```
int len(const char *s) { const char *p = s; while (*p) p++; return (int)(p - s); }
```

Answer: `len("pointer") = 7`.

Q4. What is the difference between `int *p[5]` and `int (*p)[5]`?

`int *p[5]` is an array of 5 pointers to `int`. `int (*p)[5]` is a single pointer to an array of 5 `ints` (brackets bind tighter than `*`, so parentheses change the meaning).

Answer: Array of pointers vs pointer to an array.

Q5. Rewrite `a[i][j]` using pointer notation.

`a[i][j] = *(a[i] + j) = *(*a + i) + j`.

Answer: `*(*a + i) + j`.

Q6. Write code to grow a dynamically allocated array safely with `realloc`.

```
int *tmp = realloc(a, newn * sizeof *a); if (tmp == NULL) { /* handle error; a is still valid */ } else a = tmp;
```

Answer: Always assign `realloc`'s result to a temporary pointer first.

Q7. What will happen: `int *p = NULL; printf("%d", *p);`

Dereferencing a `NULL` pointer is undefined behaviour; on most systems the program crashes (segmentation fault).

Answer: Run-time crash (segmentation fault).

Q8. How many bytes does `malloc(10 * sizeof(double))` allocate?

10×8 .

Answer: 80 bytes.

3.12 MCQs: Module III

1. A pointer is a variable that stores

- (a) a function (b) a character (c) an address (d) a value
☐ (c) — Definition.

2. The `&` operator gives

- (a) AND of bits only (b) value at address (c) address of a variable (d) reference type
☐ (c) — Address-of.

3. The `*` operator applied to a pointer gives

- (a) the value at the stored address (b) `NULL` (c) its address (d) its size
☐ (a) — Dereference.

4. If `int x = 10, *p = &x;`, the value of `*p` is

- (a) garbage (b) 10 (c) address of x (d) 0
☐ (b) — Points to x.

5. If `int *p` holds 1000 (4-byte int), `p + 1` equals

- (a) 1008 (b) 1002 (c) 1004 (d) 1001
☐ (c) — Scaled by `sizeof(int)`.

6. If `char *c` holds 1000, `c + 1` equals

- (a) 1002 (b) 1001 (c) 1004 (d) 1008
☐ (b) — `sizeof(char) = 1`.



- 7.** Which operation on pointers is not allowed?
(a) comparing pointers (b) adding two pointers (c) adding an integer (d) subtracting two pointers of the same array
☐ (b) — Meaningless.
- 8.** The array name `a` is equivalent to
(a) `&a[0]` (b) `*a` (c) `a[0]` (d) `&a[1]`
☐ (a) — Base address.
- 9.** `a[i]` is equivalent to
(a) `&a + i` (b) `*(a + i)` (c) `a + *i` (d) `*a + i`
☐ (b) — Pointer arithmetic.
- 10.** A pointer that has not been initialised is called
(a) NULL pointer (b) void pointer (c) dangling pointer (d) wild pointer
☐ (d) — Contains garbage.
- 11.** A pointer pointing to freed memory is called
(a) dangling (b) wild (c) NULL (d) generic
☐ (a) — Memory no longer valid.
- 12.** A void pointer
(a) is always NULL (b) cannot store addresses (c) can store the address of any data type (d) is a function pointer
☐ (c) — Generic pointer.
- 13.** The size of a pointer on a 64-bit system is typically
(a) 4 bytes (b) depends on the data type pointed to (c) 8 bytes (d) 2 bytes
☐ (c) — All pointers same size.
- 14.** `int **pp;` declares
(a) a pointer to a pointer to int (b) an array (c) a pointer to int (d) two pointers
☐ (a) — Double indirection.
- 15.** The operator used to access a member through a structure pointer is
(a) `->` (b) `*` (c) `.` (d) `::`
☐ (a) — `p->x` $\equiv$ `(*p).x`.
- 16.** In call by reference, the function receives
(a) constants (b) copies of values (c) nothing (d) addresses of variables
☐ (d) — Pointers.
- 17.** To swap two variables using a function, we must use
(a) call by reference (pointers) (b) call by value (c) global variables only (d) recursion
☐ (a) — Modify caller's variables.
- 18.** `malloc()` returns
(a) `char *` (b) NULL always (c) `int *` (d) `void *`
☐ (d) — Generic pointer; cast if needed.
- 19.** If `malloc()` fails, it returns
(a) `-1` (b) 0 bytes (c) a random address (d) NULL
☐ (d) — Always check.
- 20.** `calloc()` initialises allocated memory to
(a) one (b) zero (c) NULL characters only (d) garbage
☐ (b) — Zero-filled.
- 21.** The function used to change the size of allocated memory is
(a) `calloc` (b) `realloc` (c) `free` (d) `malloc`
☐ (b) — Resize.



- 22.** Memory allocated dynamically is released using
(a) remove() (b) delete (c) free() (d) dealloc()
□ (c) — stdlib.h.
- 23.** Dynamic memory is allocated from the
(a) code segment (b) register (c) heap (d) stack
□ (c) — Heap.
- 24.** A memory leak occurs when
(a) memory is freed twice (b) the stack overflows (c) a pointer is NULL (d) allocated memory is never freed and its address is lost
□ (d) — Lost memory.
- 25.** The header file for malloc() is
(a) string.h (b) malloc.c (c) stdio.h (d) stdlib.h
□ (d) — Standard library.
- 26.** calloc(5, sizeof(int)) allocates (4-byte int)
(a) 9 bytes (b) 5 bytes (c) 4 bytes (d) 20 bytes
□ (d) — 5×4 .
- 27.** *p++ is equivalent to
(a) (*p)++ (b) *++p (c) ++*p (d) *(p++)
□ (d) — Postfix binds tighter.
- 28.** (*p)++
(a) increments the value pointed to (b) is illegal (c) dereferences twice (d) moves the pointer
□ (a) — Parentheses first.
- 29.** The difference of two pointers to elements of the same array gives
(a) an address (b) zero always (c) the number of elements between them (d) the number of bytes
□ (c) — Scaled.
- 30.** char *s = "hello"; s[0] = 'H'; results in
(a) undefined behaviour (string literal is read-only) (b) "hello" (c) "Hello" (d) compile error always
□ (a) — Use char s[] instead.
- 31.** int (*fp)(int, int); declares
(a) a function returning int pointer (b) an int (c) a pointer to a function taking two ints and returning int (d) an array of pointers
□ (c) — Function pointer.
- 32.** Which is correct to read an integer into a[i] using pointers?
(a) scanf("%d", &a) (b) scanf("%d", a[i]) (c) scanf("%d", *(a + i)) (d) scanf("%d", a + i)
□ (d) — Needs an address.
- 33.** A function that returns the address of a local variable causes
(a) a dangling pointer (b) a correct result (c) compile error always (d) a memory leak
□ (a) — Local destroyed on return.
- 34.** The NULL pointer value is defined in
(a) conio.h (b) stdio.h and stdlib.h (and others) (c) math.h only (d) none
□ (b) — Macro NULL.
- 35.** Static memory allocation happens at
(a) load time only (b) run time (c) link time only (d) compile time
□ (d) — Fixed size.



36. The benefit of dynamic memory allocation is

- (a) no need to free (b) faster than stack always (c) memory can be allocated as required at run time (d) fixed size

☐ (c) — Flexibility.

37. `realloc(ptr, 0)` generally

- (a) causes error always (b) does nothing (c) doubles memory (d) may free the memory (implementation-defined)

☐ (d) — Avoid; use `free`.

38. If `p` is an `int` pointer, `sizeof(*p)` is

- (a) size of `int` (b) 0 (c) 1 (d) size of pointer

☐ (a) — Pointed-to type.

39. Array of pointers is useful for

- (a) file handling only (b) storing strings of different lengths efficiently (c) sorting only numbers (d) recursion

☐ (b) — Ragged array.

40. `free(p); p = NULL;` is good practice because

- (a) it prevents using a dangling pointer (b) it is required by the compiler (c) it frees memory twice (d) it allocates new memory

☐ (a) — Safety.

Module IV: Data Structures (Linked List, Stack, Queue)

4.1 Introduction to data structures

A **data structure** is a way of organising and storing data in memory so that it can be used efficiently. The choice of data structure affects the speed and memory needs of a program.

Classification.

- **Primitive:** int, float, char, pointer (directly supported by the machine).
- **Non-primitive:**
 - **Linear:** elements form a sequence: arrays, linked lists, stacks, queues.
 - **Non-linear:** elements form a hierarchy or network: trees, graphs.
- **Static** (fixed size, e.g. arrays) vs **dynamic** (size changes at run time, e.g. linked lists).

Basic operations: traversal, insertion, deletion, searching, sorting, merging.

Abstract data type (ADT): a data type defined by its operations and their behaviour, not by its implementation (e.g. a stack is defined by push, pop and peek; it can be implemented with an array or a linked list).

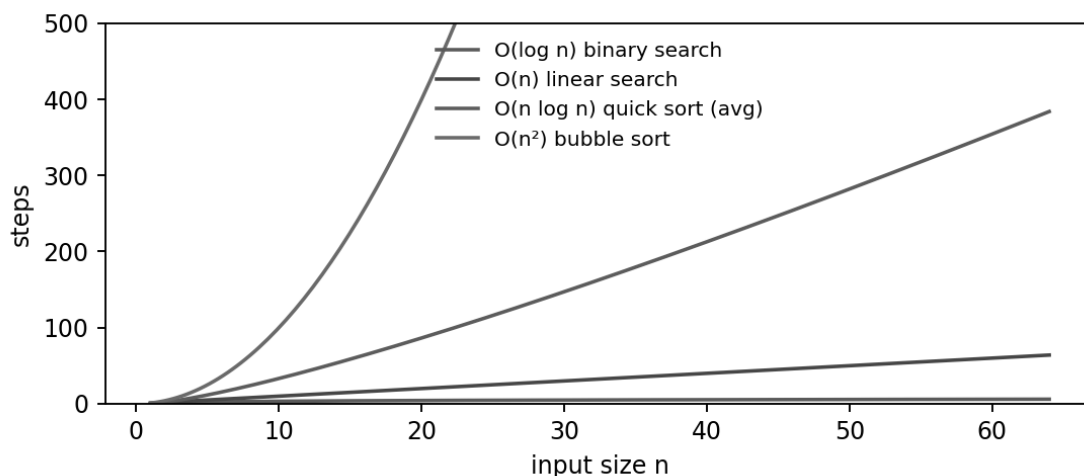


Figure 4.2: Growth of common running times. $O(\log n)$ grows very slowly; $O(n^2)$ grows fastest.

Algorithm efficiency (Big-O notation). The running time as a function of input size n : $O(1)$ constant, $O(\log n)$ logarithmic (binary search), $O(n)$ linear (linear search), $O(n \log n)$ (quick sort average), $O(n^2)$ quadratic (bubble sort).

4.2 Linked list

A **linked list** is a linear collection of **nodes**, where each node contains **data** and a **pointer (link)** to the next node. The first node is pointed to by a **head** (start) pointer; the last node's link is **NULL**.

```
struct node {  
    int data;  
    struct node *next;    /* self-referential structure */  
};
```

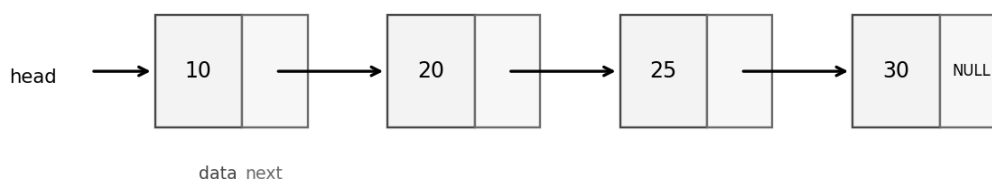


Figure 4.1: A singly linked list. Each node has a data field and a next pointer; the last node points to NULL.

Array	Linked list
Fixed size (static)	Dynamic size; grows and shrinks at run time
Contiguous memory	Nodes scattered in memory, connected by pointers
Direct (random) access: $a[i]$ in $O(1)$	Sequential access only: $O(n)$ to reach the i -th node
Insertion/deletion needs shifting: $O(n)$	Insertion/deletion by changing pointers: $O(1)$ once the position is found
No extra memory per element	Extra memory for a pointer in each node

Types of linked lists.

- **Singly linked list:** each node points to the next node only.
- **Doubly linked list:** each node has two pointers, prev and next; can be traversed in both directions.
- **Circular linked list:** the last node points back to the first node.

Operations on a singly linked list.

- **Creation / traversal:** start at head and follow next pointers until NULL.
- **Insertion at the beginning:** $new \rightarrow next = head$; $head = new$;
- **Insertion at the end:** traverse to the last node; $last \rightarrow next = new$; $new \rightarrow next = NULL$.
- **Insertion after a given node (position):** $new \rightarrow next = p \rightarrow next$; $p \rightarrow next = new$;
- **Deletion of the first node:** $temp = head$; $head = head \rightarrow next$; $free(temp)$;
- **Deletion of a given value:** find the node before it (prev); $prev \rightarrow next = cur \rightarrow next$; $free(cur)$;
- **Search:** traverse and compare.

Complete program: singly linked list with insertion and deletion.



```
#include <stdio.h>
#include <stdlib.h>
struct node { int data; struct node *next; };

struct node *newNode(int x)
{
    struct node *t = malloc(sizeof *t);
    if (!t) { printf("Overflow\n"); exit(1); }
    t->data = x; t->next = NULL;
    return t;
}
void insertBegin(struct node **head, int x)
{
    struct node *t = newNode(x);
    t->next = *head;
    *head = t;
}
void insertEnd(struct node **head, int x)
{
    struct node *t = newNode(x), *p = *head;
    if (*head == NULL) { *head = t; return; }
    while (p->next != NULL) p = p->next;
    p->next = t;
}
void insertAfterPos(struct node *head, int pos, int x) /* pos starts at 1 */
{
    struct node *p = head, *t;
    int i;
    for (i = 1; i < pos && p != NULL; i++) p = p->next;
    if (p == NULL) { printf("Invalid position\n"); return; }
    t = newNode(x);
    t->next = p->next;
    p->next = t;
}
void deleteValue(struct node **head, int x)
{
    struct node *cur = *head, *prev = NULL;
    while (cur != NULL && cur->data != x) { prev = cur; cur = cur->next; }
    if (cur == NULL) { printf("%d not found\n", x); return; }
    if (prev == NULL) *head = cur->next; /* deleting first node */
}
```

```
    else prev->next = cur->next;
    free(cur);
}
void display(struct node *p)
{
    while (p != NULL) { printf("%d -> ", p->data); p = p->next; }
    printf("NULL\n");
}
int main(void)
{
    struct node *head = NULL;
    insertEnd(&head, 20);
    insertEnd(&head, 30);
    insertBegin(&head, 10);
    insertAfterPos(head, 2, 25);
    display(head);
    deleteValue(&head, 10);
    deleteValue(&head, 30);
    display(head);
    while (head) { struct node *t = head; head = head->next; free(t); }
    return 0;
}
```

Output: 10 -> 20 -> 25 -> 30 -> NULL / 20 -> 25 -> NULL.

Why struct node **head? Functions that may change the head pointer (insertion at the beginning, deleting the first node) need the address of the head pointer (pointer to pointer, Module III), otherwise the change would be made only to a local copy.

Doubly linked list node: struct dnode { int data; struct dnode *prev, *next; }; Insertion after p: t->next = p->next; t->prev = p; if (p->next) p->next->prev = t; p->next = t;

Applications of linked lists: implementing stacks, queues and graphs (adjacency lists); polynomial arithmetic; dynamic memory management; undo functionality; music playlists (circular lists).

4.3 Stack

A **stack** is a linear data structure in which insertion and deletion take place at only one end, called the **top**. It follows **LIFO** (Last In, First Out): the element inserted last is removed first. Example: a pile of plates.

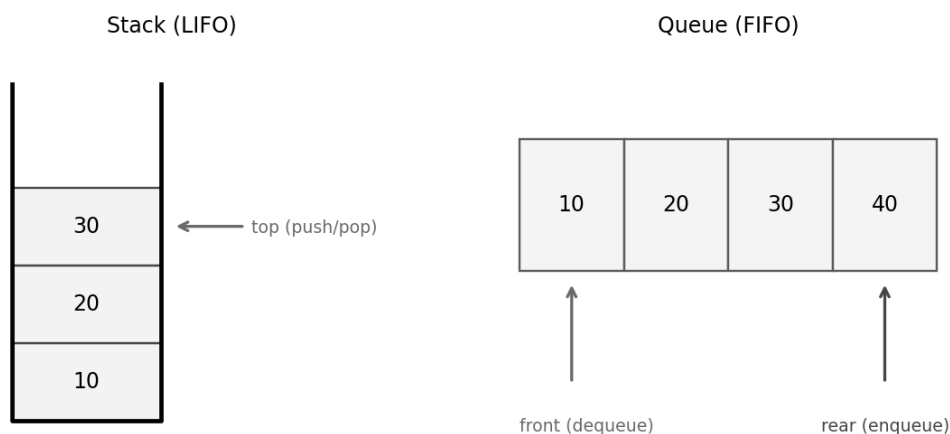


Figure 4.3: A stack (insertion and deletion at the top) and a queue (insertion at the rear, deletion at the front).

Operations.

- **push(x)**: insert x at the top. If the stack is full → **overflow**.
- **pop()**: remove and return the top element. If the stack is empty → **underflow**.
- **peek()/top()**: return the top element without removing it.
- **isEmpty(), isFull()**.

Array implementation: an array stack[MAX] and an integer top, initially -1. push: stack[++top] = x (if top < MAX - 1). pop: return stack[top--] (if top ≥ 0). All operations are O(1).

```
#include <stdio.h>
#define MAX 5
int stack[MAX], top = -1;
void push(int x)
{
    if (top == MAX - 1) { printf("Stack Overflow\n"); return; }
    stack[++top] = x;
}
int pop(void)
{
    if (top == -1) { printf("Stack Underflow\n"); return -1; }
    return stack[top--];
}
void show(void)
{
    int i;
    printf("Stack (top first): ");
    for (i = top; i >= 0; i--) printf("%d ", stack[i]);
    printf("\n");
}
int main(void)
{
    push(10); push(20); push(30);
    show();
    printf("Popped %d\n", pop());
    show();
    return 0;
}
```

Output: Stack (top first): 30 20 10 / Popped 30 / Stack (top first): 20 10.

Linked-list implementation: push = insert at the beginning; pop = delete the first node. No fixed size limit.

Applications of stacks: function calls and recursion (call stack); expression conversion and evaluation; parenthesis matching; undo/redo in editors; back button in browsers; reversing a string; depth-first search.

4.4 Arithmetic expression notations

- **Infix:** operator between operands: A + B. Needs precedence rules and parentheses.
- **Prefix (Polish):** operator before operands: + A B.
- **Postfix (Reverse Polish):** operator after operands: A B +. No parentheses or precedence rules are needed, and it can be evaluated in one left-to-right scan using a stack; compilers convert infix expressions to postfix.

Infix	Prefix	Postfix
A + B * C	+ A * B C	A B C * +



Infix	Prefix	Postfix
$(A + B) * C$	$* + A B C$	$A B + C *$
$A * B + C / D$	$+ * A B / C D$	$A B * C D / +$
$(A + B) * (C - D)$	$* + A B - C D$	$A B + C D - *$
$A + B - C$	$- + A B C$	$A B + C -$

4.5 Infix to postfix conversion (algorithm)

Scan the infix expression from left to right, using a stack for operators:

- **Operand:** add it to the postfix output.
- **'(':** push it onto the stack.
- **')':** pop and output operators until '(' is found; discard the '('.
- **Operator op:** while the stack top is an operator with **higher precedence, or equal precedence and left associativity**, pop it to the output; then push op. (Precedence: ^ highest, then * / %, then + -. ^ is right-associative.)
- **End of expression:** pop all remaining operators to the output.

Worked example: convert $A + B * (C - D) / E$ to postfix.

Symbol	Stack	Postfix output
A	(empty)	A
+	+	A
B	+	A B
*	+ *	A B
(	+ * (	A B
C	+ * (	A B C
-	+ * (-	A B C
D	+ * (-	A B C D
)	+ *	A B C D -
/	+ /	A B C D - *
E	+ /	A B C D - * E
end	(empty)	A B C D - * E / +

Result: **A B C D - * E / +**. (When '/' arrived, '*' on the stack had equal precedence and left associativity, so it was popped first.)

Program: infix to postfix (single-character operands).

```
#include <stdio.h>
#include <ctype.h>
char st[100]; int top = -1;
void push(char c) { st[++top] = c; }
char pop(void) { return st[top--]; }
int prec(char c)
{
    if (c == '^') return 3;
    if (c == '*' || c == '/' || c == '%') return 2;
    if (c == '+' || c == '-') return 1;
    return 0;
}
int main(void)
{
    char in[100], out[100], *e, c;
    int k = 0;
    printf("Infix: ");
    scanf("%99s", in);
    for (e = in; *e; e++) {
        if (isalnum((unsigned char)*e)) out[k++] = *e;
        else if (*e == '(') push(*e);
        else if (*e == ')') { while ((c = pop()) != '(') out[k++] = c; }
        else {
            while (top != -1 && (prec(st[top]) > prec(*e) ||
                (prec(st[top]) == prec(*e) && *e != '^')))
                out[k++] = pop();
            push(*e);
        }
    }
    while (top != -1) out[k++] = pop();
    out[k] = '\0';
    printf("Postfix: %s\n", out);
    return 0;
}
```

Input $A+B*(C-D)/E \rightarrow$ Postfix: $ABCD-*E/+$.

4.6 Evaluation of a postfix expression

Scan from left to right:

- **Operand:** push it onto the stack.
- **Operator:** pop the top two elements (first popped = right operand B, second = left operand A), compute $A \text{ op } B$, and push the result.
- At the end, the single value on the stack is the result.

Worked example: evaluate $6\ 2\ 3\ +\ -\ 3\ 8\ 2\ /\ +\ *$ (infix: $(6 - (2 + 3)) * (3 + 8/2)$).

Symbol	Action	Stack
6, 2, 3	push	6 2 3
+	$2 + 3 = 5$	6 5
-	$6 - 5 = 1$	1
3, 8, 2	push	1 3 8 2
/	$8 / 2 = 4$	1 3 4
+	$3 + 4 = 7$	1 7
*	$1 * 7 = 7$	7

Result: 7.

Program: postfix evaluation (single-digit operands).

```
#include <stdio.h>
#include <ctype.h>
int st[100], top = -1;
int main(void)
{
    char exp[100], *e;
    int a, b;
    printf("Postfix: ");
    scanf("%99s", exp);
    for (e = exp; *e; e++) {
        if (isdigit((unsigned char)*e)) st[++top] = *e - '0';
        else {
            b = st[top--]; a = st[top--];
            switch (*e) {
                case '+': st[++top] = a + b; break;
                case '-': st[++top] = a - b; break;
                case '*': st[++top] = a * b; break;
                case '/': st[++top] = a / b; break;
            }
        }
    }
    printf("Value = %d\n", st[top]);
    return 0;
}
```

Input 623+-382/+* → Value = 7.

4.7 Queue

A **queue** is a linear data structure in which insertion takes place at one end (**rear**) and deletion at the other end (**front**). It follows **FIFO** (First In, First Out). Example: people waiting in a line at a ticket counter.

Operations. enqueue(x) (insert at rear; overflow if full), dequeue() (delete from front; underflow if empty), peek/front, isEmpty, isFull.

Linear queue using an array. front = rear = -1 initially.

- enqueue: if rear == MAX - 1 → overflow; else { if (front == -1) front = 0; q[++rear] = x; }
- dequeue: if front == -1 or front > rear → underflow; else x = q[front++].

Drawback of the linear queue: after some deletions, the empty positions at the front cannot be reused. The queue may report "overflow" even though there is space at the front.

Circular queue removes this drawback by treating the array as circular: after the last position comes the first. Indices move with modulo arithmetic:

```
rear = (rear + 1) % MAX; front = (front + 1) % MAX
```

```
Full: (rear + 1) % MAX == front; Empty: front == -1 (in the version below)
```

```
#include <stdio.h>
#define MAX 5
int q[MAX], front = -1, rear = -1;
void enqueue(int x)
{
    if ((rear + 1) % MAX == front) { printf("Queue Overflow\n"); return; }
    if (front == -1) front = 0;
    rear = (rear + 1) % MAX;
    q[rear] = x;
}
int dequeue(void)
{
    int x;
    if (front == -1) { printf("Queue Underflow\n"); return -1; }
    x = q[front];
    if (front == rear) front = rear = -1;          /* queue becomes empty */
    else front = (front + 1) % MAX;
    return x;
}
void display(void)
{
    int i;
    if (front == -1) { printf("Queue empty\n"); return; }
    printf("Queue: ");
    for (i = front; ; i = (i + 1) % MAX) {
        printf("%d ", q[i]);
        if (i == rear) break;
    }
    printf("\n");
}
int main(void)
{
    int i;
    for (i = 1; i <= 5; i++) enqueue(i * 10);    /* 10..50, queue full */
    enqueue(60);                                /* overflow */
    printf("Dequeued %d\n", dequeue());
    printf("Dequeued %d\n", dequeue());
    enqueue(60); enqueue(70);                    /* reuse freed slots */
    display();
    return 0;
}
```

Output: Queue Overflow / Dequeued 10 / Dequeued 20 / Queue: 30 40 50 60 70.

Linked-list queue: enqueue inserts at the end (maintain a rear pointer), dequeue deletes from the beginning; no fixed size.

Other queues (awareness): deque (double-ended queue: insertion and deletion at both ends); **priority queue** (elements removed in order of priority, not arrival).

Stack	Queue
LIFO	FIFO
Insertion and deletion at one end (top)	Insertion at rear, deletion at front
Operations: push, pop	Operations: enqueue, dequeue
One pointer (top)	Two pointers (front, rear)
Recursion, expression evaluation, undo	CPU scheduling, printer spooling, buffering, BFS

Applications of queues: CPU and process scheduling (round robin uses a circular queue), printer and disk queues, keyboard buffer, call centre systems, breadth-first search, data packets in networks.

4.8 Common mistakes to avoid in Module IV

- Forgetting to check for overflow/underflow before push/pop or enqueue/dequeue.
- Losing the rest of a linked list by changing head before saving the next pointer.
- Not freeing deleted nodes (memory leak).
- Forgetting that the second popped operand is the left operand in postfix evaluation ($A - B$, not $B - A$).
- Using $(\text{rear} + 1)$ instead of $(\text{rear} + 1) \% \text{MAX}$ in a circular queue.

4.9 Solved problems: Module IV

Problem 1. Convert the infix expression $(A + B) * C - (D - E) * (F + G)$ to postfix, showing the stack.

$(A + B) \rightarrow A B +$; then $* C \rightarrow A B + C *$; $-$ is lower than $*$ so $*$ is output before $-$.

$(D - E) \rightarrow D E -$; $(F + G) \rightarrow F G +$; their product: $D E - F G + *$.

Final: $A B + C * D E - F G + * -$

Answer: $A B + C * D E - F G + * -$

Problem 2. Convert $A \wedge B \wedge C * D$ to postfix ($\wedge$ is right-associative).

$A \wedge (B \wedge C) \rightarrow A B C \wedge \wedge$; then $* D$.

Answer: $A B C \wedge \wedge D *$

Problem 3. Convert $A * (B + C) / D - E$ to prefix and postfix.

Postfix: $A B C + * D / E -$. Prefix: $- / * A + B C D E$.

Answer: Postfix $A B C + * D / E -$; prefix $- / * A + B C D E$.

Problem 4. Evaluate the postfix expression $5\ 3\ +\ 8\ 2\ -\ *$ (show the stack).

push 5, 3; $+$ $\rightarrow$ 8; push 8, 2; $-$ $\rightarrow$ 6; stack 8 6; $*$ $\rightarrow$ 48.

Answer: 48

Problem 5. Evaluate $2\ 3\ 1\ * + 9 -$.

$3 * 1 = 3$; $2 + 3 = 5$; $5 - 9 = -4$.

Answer: -4

Problem 6. Evaluate the prefix expression $+ * 2\ 3\ 4$.

Scan prefix from right to left (push operands; on an operator, pop A then B and compute $A \text{ op } B$): 4, 3, 2 pushed; $*$ $\rightarrow 2 \times 3 = 6$; $+$ $\rightarrow 6 + 4 = 10$.

Answer: 10

Problem 7. The elements 10, 20, 30, 40 are pushed onto a stack and then two pops are done. What is on top and what has been popped?

LIFO: pops return 40, then 30. Remaining: 10 20 (top = 20).

Answer: Popped 40 and 30; top is 20.

Problem 8. A circular queue of size 5 has front = 3 and rear = 1. How many elements does it contain? Where will the next element be inserted?

Elements occupy positions 3, 4, 0, 1 → 4 elements. Next rear = $(1 + 1) \% 5 = 2$.

Answer: 4 elements; next insertion at index 2 (after which the queue is full).

Problem 9. Write a function to count the nodes of a singly linked list and another to search for a value.

```
#include <stdio.h>
#include <stdlib.h>
struct node { int data; struct node *next; };
int count(struct node *p) { int c = 0; while (p) { c++; p = p->next; } return c; }
int search(struct node *p, int key)
{
    int pos = 1;
    while (p) { if (p->data == key) return pos; p = p->next; pos++; }
    return -1;
}
int main(void)
{
    struct node c3 = {30, NULL}, c2 = {20, &c3}, c1 = {10, &c2};
    printf("Nodes = %d, 30 found at position %d\n", count(&c1), search(&c1, 30));
    return 0;
}
```

ANS: Nodes = 3, 30 found at position 3.

Problem 10. Write a function to reverse a singly linked list.

```
#include <stdio.h>
struct node { int data; struct node *next; };
struct node *reverse(struct node *head)
{
    struct node *prev = NULL, *cur = head, *nxt;
    while (cur) { nxt = cur->next; cur->next = prev; prev = cur; cur = nxt; }
    return prev;
}
int main(void)
{
    struct node c3 = {3, NULL}, c2 = {2, &c3}, c1 = {1, &c2}, *h = reverse(&c1);
    for (; h; h = h->next) printf("%d ", h->data);
    printf("\n");
    return 0;
}
```

ANS: Prints 3 2 1.

Problem 11. Implement a stack using a linked list (push and pop).

```
#include <stdio.h>
#include <stdlib.h>
struct node { int data; struct node *next; };
struct node *top = NULL;
void push(int x) { struct node *t = malloc(sizeof *t); t->data = x; t->next = top; top = t; }
int pop(void)
{
    struct node *t = top; int x;
    if (!t) { printf("Underflow\n"); return -1; }
    x = t->data; top = t->next; free(t);
    return x;
}
int main(void)
{
    push(1); push(2); push(3);
    printf("%d ", pop()); printf("%d ", pop()); printf("%d\n", pop());
    pop();
    return 0;
}
```

ANS: 3 2 1, then "Underflow".

Problem 12. Write a program to check whether the parentheses in an expression are balanced using a stack.

```
#include <stdio.h>
int main(void)
{
    char s[100], st[100]; int top = -1, i, ok = 1;
    printf("Expression: ");
    scanf("%99s", s);
    for (i = 0; s[i] && ok; i++) {
        if (s[i] == '(' || s[i] == '[' || s[i] == '{') st[++top] = s[i];
        else if (s[i] == ')' || s[i] == ']' || s[i] == '}') {
            if (top == -1) ok = 0;
            else {
                char o = st[top--];
                if ((s[i] == ')') && o != '(') || (s[i] == ']') && o != '[') || (s[i] == '}') && o != '{')
                    ok = 0;
            }
        }
    }
    printf("%s\n", ok && top == -1 ? "Balanced" : "Not balanced");
    return 0;
}
```

ANS: {a+(b*c)} → Balanced; (a+b) → Not balanced.

4.10 Practice problems with solutions: Module IV

Q1. Convert $A + B * C - D / E$ to postfix.

$B * C \rightarrow B C *$; $A + \dots \rightarrow A B C * +$; $D / E \rightarrow D E /$; then $-$.

Answer: $A B C * + D E / -$

Q2. Convert $(A - B) / (C + D) * E$ to postfix.

$(A - B) \rightarrow A B -$; $(C + D) \rightarrow C D +$; / then * left to right.

Answer: $A B - C D + / E *$



Q3. Evaluate the postfix expression $4\ 5\ *\ 6\ 2\ /\ -$.

$4 \times 5 = 20$; $6/2 = 3$; $20 - 3 = 17$.

Answer: 17

Q4. Evaluate $9\ 3\ /\ 2\ 4\ *\ +\ 1\ -$.

$9/3 = 3$; $2 \times 4 = 8$; $3 + 8 = 11$; $11 - 1 = 10$.

Answer: 10

Q5. A stack of size 4 receives: push 1, push 2, pop, push 3, push 4, push 5, push 6. What happens?

After push 1, push 2, pop: [1]. push 3, 4, 5: [1, 3, 4, 5] (full). push 6 → overflow.

Answer: Stack becomes 1 3 4 5 (top = 5); pushing 6 causes overflow.

Q6. In a linear queue of size 5, after inserting 5 elements and deleting 3, can a new element be inserted? What about a circular queue?

Linear: $\text{rear} = 4 = \text{MAX} - 1$, so it reports overflow even though 3 slots are free.

Circular: $(4 + 1) \% 5 = 0 \neq \text{front} (3)$, so insertion succeeds at index 0.

Answer: No for a linear queue; yes for a circular queue.

Q7. Write the steps to delete the last node of a singly linked list.

If the list has one node: free it, head = NULL. Otherwise traverse with p until $p \rightarrow \text{next} \rightarrow \text{next} == \text{NULL}$; $\text{free}(p \rightarrow \text{next})$; $p \rightarrow \text{next} = \text{NULL}$.

Answer: As described ($O(n)$ time).

Q8. Write the structure of a node of a doubly linked list and the steps to delete a node p from the middle.

```
struct dnode { int data; struct dnode *prev, *next; };  
p->prev->next = p->next;  
p->next->prev = p->prev; free(p);
```

Answer: As above.

Q9. Which data structure is used to (a) evaluate postfix expressions, (b) schedule print jobs, (c) implement recursion, (d) implement BFS?

(a) stack; (b) queue; (c) stack; (d) queue.

Answer: Stack, queue, stack, queue.

4.11 MCQs: Module IV

1. A stack follows

(a) priority order (b) LIFO (c) random order (d) FIFO

□ (b) — Last in, first out.

2. A queue follows

(a) LIFO (b) LILO only (c) random (d) FIFO

□ (d) — First in, first out.



- 3.** Inserting an element into a full stack causes
(a) deletion (b) nothing (c) overflow (d) underflow
☐ (c) — No space.
- 4.** Deleting from an empty queue causes
(a) nothing (b) overflow (c) deadlock (d) underflow
☐ (d) — Nothing to remove.
- 5.** In an array stack, the initial value of top is usually
(a) 1 (b) 0 (c) -1 (d) MAX
☐ (c) — Empty stack.
- 6.** Which data structure is used for function calls and recursion?
(a) graph (b) stack (c) queue (d) tree
☐ (b) — Call stack.
- 7.** The postfix form of $A + B * C$ is
(a) $A B C + *$ (b) $A B + C *$ (c) $A B C * +$ (d) $+ A * B C$
☐ (c) — $*$ has higher precedence.
- 8.** The postfix form of $(A + B) * C$ is
(a) $A B C + *$ (b) $* + A B C$ (c) $A B + C *$ (d) $A + B C *$
☐ (c) — Parentheses first.
- 9.** The prefix form of $A * B + C$ is
(a) $A B * C +$ (b) $* A + B C$ (c) $+ A * B C$ (d) $+ * A B C$
☐ (d) — Operator first.
- 10.** The value of postfix $2\ 3\ 4\ *\ +$ is
(a) 24 (b) 14 (c) 20 (d) 9
☐ (b) — $3 \times 4 + 2$.
- 11.** In postfix evaluation, when an operator is read,
(a) all are popped (b) nothing is popped (c) two operands are popped (d) one operand is popped
☐ (c) — Binary operator.
- 12.** Which notation needs no parentheses?
(a) postfix (b) infix (c) both (d) neither
☐ (a) — Order implied.
- 13.** In a linked list, each node contains
(a) only address (b) only data (c) data and a link to the next node (d) an array
☐ (c) — Self-referential structure.
- 14.** The last node of a singly linked list points to
(a) NULL (b) previous node (c) itself (d) head
☐ (a) — End marker.
- 15.** In a circular linked list, the last node points to
(a) itself always (b) the first node (c) NULL (d) the middle node
☐ (b) — Circle.
- 16.** A doubly linked list node contains
(a) two pointers (prev and next) (b) one pointer (c) three pointers (d) none
☐ (a) — Two-way traversal.
- 17.** Accessing the i -th element of a linked list takes
(a) $O(n)$ (b) $O(\log n)$ (c) $O(1)$ (d) $O(n^2)$
☐ (a) — Sequential access.



- 18.** The main advantage of a linked list over an array is
(a) random access (b) less memory per element (c) dynamic size and easy insertion/deletion (d) cache friendliness
☐ (c) — Pointer changes only.
- 19.** A self-referential structure contains
(a) a union (b) no members (c) a pointer to a structure of the same type (d) an array of itself
☐ (c) — struct node *next.
- 20.** To insert a node at the beginning, the correct steps are
(a) free(head); (b) head->next = new; (c) new->next = head; head = new; (d) head = new; new->next = head;
☐ (c) — Order matters.
- 21.** The drawback of a linear array queue is
(a) it is slow (b) it is LIFO (c) freed front positions cannot be reused (d) it cannot store integers
☐ (c) — False overflow.
- 22.** A circular queue is full when
(a) front == rear (b) front == -1 (c) rear == MAX - 1 (d) (rear + 1) % MAX == front
☐ (d) — Standard condition.
- 23.** In a circular queue of size 6, if rear = 5, the next rear is
(a) 5 (b) 6 (c) 0 (d) 1
☐ (c) — $(5 + 1) \% 6$.
- 24.** Which data structure is used in breadth-first search?
(a) tree (b) heap only (c) queue (d) stack
☐ (c) — Level order.
- 25.** Which data structure is used for parenthesis matching?
(a) queue (b) linked list only (c) array only (d) stack
☐ (d) — LIFO matching.
- 26.** A deque allows
(a) no deletion (b) insertion and deletion at both ends (c) deletion only at front (d) insertion only at rear
☐ (b) — Double-ended queue.
- 27.** A priority queue removes elements
(a) according to priority (b) in FIFO order (c) in LIFO order (d) randomly
☐ (a) — Highest priority first.
- 28.** Stack and queue are examples of
(a) linear data structures (b) graphs (c) trees (d) non-linear data structures
☐ (a) — Sequential organisation.
- 29.** Trees and graphs are
(a) non-linear (b) static only (c) linear (d) primitive
☐ (a) — Hierarchical/network.
- 30.** The time complexity of push and pop in an array stack is
(a) $O(n^2)$ (b) $O(1)$ (c) $O(n)$ (d) $O(\log n)$
☐ (b) — Constant time.
- 31.** Round-robin CPU scheduling uses
(a) linked stack (b) stack (c) circular queue (d) tree
☐ (c) — Cyclic order.



- 32.** The undo operation in editors typically uses
(a) stack (b) graph (c) tree (d) queue
☐ (a) — Last action undone first.
- 33.** In the infix-to-postfix algorithm, ')' causes
(a) push (b) popping until '(' (c) nothing (d) popping everything
☐ (b) — Close sub-expression.
- 34.** If 1, 2, 3 are pushed in that order with no pops in between, the pop sequence is
(a) 2 3 1 (b) 1 2 3 (c) 3 1 2 (d) 3 2 1
☐ (d) — LIFO order.
- 35.** The number of pointers changed to insert a node in the middle of a singly linked list is
(a) 1 (b) 3 (c) 0 (d) 2
☐ (d) — new->next and prev->next.
- 36.** A linked list is preferred over an array when
(a) frequent insertion and deletion is needed (b) frequent random access is needed (c) memory is extremely limited (d) size is fixed
☐ (a) — Pointer operations.
- 37.** The postfix of $A - B + C$ is
(a) $A B - C +$ (b) $A B C - +$ (c) $- A + B C$ (d) $A B C + -$
☐ (a) — Left associativity.
- 38.** Stack underflow occurs when
(a) $\text{rear} == \text{front}$ (b) $\text{top} == -1$ and pop is attempted (c) $\text{top} == \text{MAX} - 1$ (d) $\text{front} == -1$ in a queue
☐ (b) — Empty stack.
- 39.** A queue implemented with a linked list needs pointers to
(a) front and rear (b) only rear (c) only front (d) middle
☐ (a) — Constant-time enqueue and dequeue.
- 40.** Which is an application of a linked list?
(a) polynomial representation (b) binary search (c) matrix multiplication only (d) none
☐ (a) — Terms as nodes.

Module V: Trees, Sorting and Searching

5.1 Trees: terminology

A **tree** is a non-linear, hierarchical data structure consisting of nodes connected by edges, with one special node called the **root**, and no cycles.

- **Root:** the topmost node (no parent).
- **Parent, child, siblings:** a node directly above another is its parent; nodes with the same parent are siblings.
- **Leaf (terminal node):** a node with no children. **Internal node:** a node with at least one child.
- **Edge:** a link between two nodes. A tree with n nodes has $n - 1$ edges.
- **Degree of a node:** number of its children. **Degree of a tree:** maximum degree of any node.
- **Level:** root is at level 0 (some books use 1); children of a node at level k are at level $k + 1$.
- **Height (depth) of a tree:** number of edges on the longest path from the root to a leaf (or number of levels, depending on convention; state yours).
- **Subtree:** a node together with all its descendants.
- **Ancestors and descendants.**

5.2 Binary tree

A **binary tree** is a tree in which each node has **at most two children**, called the left child and the right child.

```
struct tnode {  
    int data;  
    struct tnode *left, *right;  
};
```

Types of binary trees.

- **Strictly (full) binary tree:** every node has either 0 or 2 children.
- **Complete binary tree:** all levels are completely filled except possibly the last, which is filled from the left.
- **Perfect binary tree:** all internal nodes have 2 children and all leaves are at the same level.
- **Skewed binary tree:** every node has only one child (left-skewed or right-skewed); behaves like a linked list.

Properties.

Maximum nodes at level $k = 2^k$ (root at level 0); maximum nodes in a tree of height $h = 2^{h+1} - 1$

Minimum height of a binary tree with n nodes $= \lceil \log_2(n + 1) \rceil - 1$; in a full binary tree, leaves = internal nodes + 1

Representation. (1) **Linked** (nodes with left and right pointers; most common). (2) **Array (sequential):** root at index 0; for a node at index i , left child at $2i + 1$ and right child at $2i + 2$ (with 1-based indexing: $2i$ and $2i + 1$); parent at $(i - 1)/2$. Suitable for complete binary trees (heaps).

5.3 Tree traversals

Traversal means visiting every node exactly once. For a binary tree with root N, left subtree L and right subtree R:

Traversal	Order	Rule
Preorder	N L R	Visit root, then left subtree, then right subtree
Inorder	L N R	Left subtree, root, right subtree
Postorder	L R N	Left subtree, right subtree, root
Level order	Level by level, left to right	Uses a queue (breadth-first)

Example. For the tree with root A, A's children B (left) and C (right), B's children D and E, and C's right child F:

- Preorder: A B D E C F
- Inorder: D B E A C F
- Postorder: D E B F C A
- Level order: A B C D E F

Recursive traversal functions.

```
void preorder(struct tnode *r) { if (r) { printf("%d ", r->data); preorder(r->left); preorder(r->right); } }
void inorder(struct tnode *r) { if (r) { inorder(r->left); printf("%d ", r->data); inorder(r->right); } }
void postorder(struct tnode *r) { if (r) { postorder(r->left); postorder(r->right); printf("%d ", r->data); } }
```

Expression trees. An arithmetic expression can be stored as a binary tree with operators at internal nodes and operands at leaves. Preorder gives the prefix form, inorder gives infix, and postorder gives postfix. Example: $(A + B) * C$ has root *, left subtree $+$ (A, B), right leaf C; postorder $A B + C *$.

Constructing a binary tree from traversals. A unique binary tree can be built from inorder + preorder (or inorder + postorder). The first element of preorder (last of postorder) is the root; locate it in inorder to split into left and right subtrees; repeat.

5.4 Binary search tree (BST)

A **binary search tree** is a binary tree in which, for every node:

- all values in the **left subtree are smaller** than the node's value, and
- all values in the **right subtree are larger** (duplicates are either not allowed or placed consistently on one side).

Key property: the inorder traversal of a BST gives the values in **ascending (sorted) order**.

Search: start at the root; if key == node, found; if key < node, go left; else go right; if NULL is reached, not found. Time $O(h)$: $O(\log n)$ for a balanced tree, $O(n)$ for a skewed tree.

Insertion: search for the key; insert the new node at the NULL position where the search ends (always as a leaf).

Deletion (three cases):

- **Leaf node:** simply remove it.
- **Node with one child:** replace the node by its child.

- **Node with two children:** replace its value with its **inorder successor** (smallest value in the right subtree) or **inorder predecessor** (largest in the left subtree), then delete that successor node (which has at most one child).

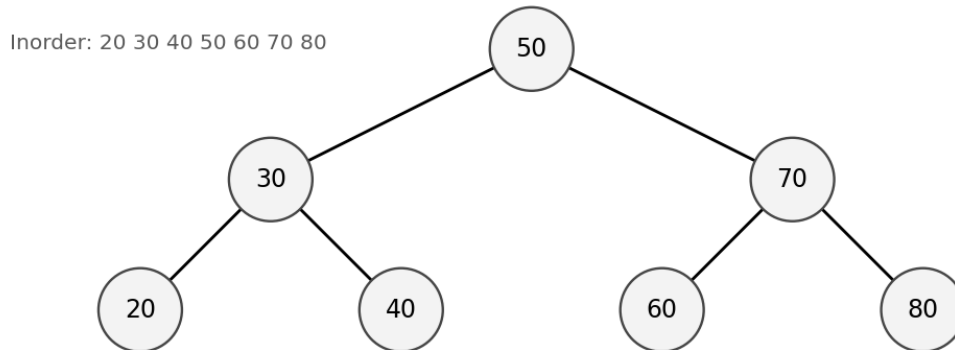


Figure 5.1: Binary search tree built from 50, 30, 70, 20, 40, 60, 80.

Example: insert 50, 30, 70, 20, 40, 60, 80 into an empty BST.

- 50 is the root; 30 < 50 goes left; 70 > 50 goes right; 20 < 50, < 30 → left of 30; 40 → right of 30; 60 → left of 70; 80 → right of 70.
- Inorder: 20 30 40 50 60 70 80 (sorted). Preorder: 50 30 20 40 70 60 80. Postorder: 20 40 30 60 80 70 50.
- Deleting 30 (two children): its inorder successor is 40; replace 30 by 40 and delete the old 40 leaf.

Complete program: binary search tree (insert, search, traversals, delete).

```
#include <stdio.h>
#include <stdlib.h>
struct tnode { int data; struct tnode *left, *right; };

struct tnode *insert(struct tnode *r, int x)
{
    if (r == NULL) {
        r = malloc(sizeof *r);
        r->data = x; r->left = r->right = NULL;
    } else if (x < r->data) r->left = insert(r->left, x);
    else if (x > r->data) r->right = insert(r->right, x);
    return r;
    /* duplicates ignored */
}

int search(struct tnode *r, int key)
{
    while (r) {
        if (key == r->data) return 1;
        r = (key < r->data) ? r->left : r->right;
    }
    return 0;
}

struct tnode *minNode(struct tnode *r) { while (r->left) r = r->left; return r; }
struct tnode *delete(struct tnode *r, int x)
{
    struct tnode *t;
    if (r == NULL) return NULL;
    if (x < r->data) r->left = delete(r->left, x);
    else if (x > r->data) r->right = delete(r->right, x);
    else {
        if (r->left == NULL) { t = r->right; free(r); return t; }
        if (r->right == NULL) { t = r->left; free(r); return t; }
        t = minNode(r->right);
        /* inorder successor */
        r->data = t->data;
        r->right = delete(r->right, t->data);
    }
    return r;
}

void inorder(struct tnode *r) { if (r) { inorder(r->left); printf("%d ", r->data); inorder(r->right); } }
void preorder(struct tnode *r) { if (r) { printf("%d ", r->data); preorder(r->left); preorder(r->right); } }
void postorder(struct tnode *r) { if (r) { postorder(r->left); postorder(r->right); printf("%d ", r->data); } }

int height(struct tnode *r)
{
    int hl, hr;
    if (r == NULL) return -1;
    hl = height(r->left); hr = height(r->right);
    return 1 + (hl > hr ? hl : hr);
}

int main(void)
{
    int keys[] = {50, 30, 70, 20, 40, 60, 80}, i;
    struct tnode *root = NULL;
    for (i = 0; i < 7; i++) root = insert(root, keys[i]);
    printf("Inorder: "); inorder(root); printf("\n");
    printf("Preorder: "); preorder(root); printf("\n");
    printf("Postorder: "); postorder(root); printf("\n");
    printf("Height = %d, search 60: %s\n", height(root), search(root, 60) ? "found" : "not found");
    root = delete(root, 30);
    printf("After deleting 30, inorder: "); inorder(root); printf("\n");
    return 0;
}
```

Output: Inorder: 20 30 40 50 60 70 80 / Preorder: 50 30 20 40 70 60 80 / Postorder: 20 40 30 60 80 70 50 / Height = 2, search 60: found / After deleting 30, inorder: 20 40 50 60 70 80.

Applications of trees: file systems (directories), BSTs for fast searching (databases, symbol tables in compilers), expression trees, heaps for priority queues, Huffman coding (compression), decision trees in AI, routing tables.

5.5 Searching

Linear (sequential) search. Compare the key with each element from the first to the last until it is found or the list ends.

- Works on unsorted or sorted data, arrays or linked lists.
- Best case 1 comparison ($O(1)$), worst case n comparisons ($O(n)$), average $(n + 1)/2$.

Binary search. Works only on a **sorted** array. Compare the key with the middle element; if equal, found; if smaller, search the left half; if larger, search the right half; repeat until found or the range becomes empty.

- $\text{mid} = (\text{low} + \text{high})/2$ (or $\text{low} + (\text{high} - \text{low})/2$ to avoid overflow).
- Each comparison halves the search space, so the worst case is about $\log_2 n$ comparisons: $O(\log n)$. For $n = 1,000,000$, at most 20 comparisons (vs up to 1,000,000 for linear search).

Linear search	Binary search
Data need not be sorted	Data must be sorted
$O(n)$ time	$O(\log n)$ time
Works on arrays and linked lists	Needs random access (arrays)
Simple	Slightly more complex, much faster for large n

Worked example (binary search). Find 23 in {2, 5, 8, 12, 16, 23, 38, 56, 72, 91} (indices 0–9).

Step	low	high	mid	a[mid]	Action
1	0	9	4	16	$23 > 16 \rightarrow \text{low} = 5$
2	5	9	7	56	$23 < 56 \rightarrow \text{high} = 6$
3	5	6	5	23	Found at index 5

Program: linear search and binary search.

```
#include <stdio.h>
int linearSearch(const int a[], int n, int key)
{
    int i;
    for (i = 0; i < n; i++) if (a[i] == key) return i;
    return -1;
}
int binarySearch(const int a[], int n, int key)
{
    int low = 0, high = n - 1, mid;
    while (low <= high) {
        mid = low + (high - low) / 2;
        if (a[mid] == key) return mid;
        if (key < a[mid]) high = mid - 1;
        else low = mid + 1;
    }
    return -1;
}
int main(void)
{
    int a[] = {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}, n = 10;
    printf("Linear: 38 at index %d\n", linearSearch(a, n, 38));
    printf("Binary: 23 at index %d\n", binarySearch(a, n, 23));
    printf("Binary: 40 at index %d (not found)\n", binarySearch(a, n, 40));
    return 0;
}
```

Output: Linear: 38 at index 6 / Binary: 23 at index 5 / Binary: 40 at index -1 (not found).

Recursive binary search: int bs(int a[], int lo, int hi, int key) { if (lo > hi) return -1; int mid = (lo + hi)/2; if (a[mid] == key) return mid; return key < a[mid] ? bs(a, lo, mid - 1, key) : bs(a, mid + 1, hi, key); }

5.6 Sorting

Sorting arranges elements in ascending or descending order. A sorted list allows fast searching (binary search), easy detection of duplicates, and neat reports.

Terms. In-place sort (needs no extra array), stable sort (equal elements keep their relative order), internal sort (all data in memory), external sort (data on disk).

5.7 Bubble sort

Idea. Repeatedly compare adjacent elements and swap them if they are in the wrong order. After each pass, the largest remaining element "bubbles up" to its correct position at the end.

Algorithm. For pass = 1 to n - 1: for j = 0 to n - pass - 1: if a[j] > a[j + 1], swap them. If no swaps occur in a pass, the array is sorted (early exit).

Worked example: sort {5, 1, 4, 2, 8}.

Pass	Array after the pass	Comparisons
1	1 4 2 5 8	4 (8 is in place)
2	1 2 4 5 8	3
3	1 2 4 5 8 (no swaps → stop)	2

Pass 1 in detail: (5, 1) swap → 1 5 4 2 8; (5, 4) swap → 1 4 5 2 8; (5, 2) swap → 1 4 2 5 8; (5, 8) no swap.

Complexity. Comparisons in the worst case = $(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2 \Rightarrow O(n^2)$. Best case (already sorted, with the swapped flag) $O(n)$. In-place and stable. Simple but slow for large n .

```
#include <stdio.h>
void bubbleSort(int a[], int n)
{
    int pass, j, t, swapped;
    for (pass = 1; pass < n; pass++) {
        swapped = 0;
        for (j = 0; j < n - pass; j++)
            if (a[j] > a[j + 1]) { t = a[j]; a[j] = a[j + 1]; a[j + 1] = t; swapped = 1; }
        if (!swapped) break;
    }
}
int main(void)
{
    int a[] = {5, 1, 4, 2, 8}, i;
    bubbleSort(a, 5);
    for (i = 0; i < 5; i++) printf("%d ", a[i]);
    printf("\n");
    return 0;
}
```

Output: 1 2 4 5 8.

5.8 Quick sort

Idea (divide and conquer, by C.A.R. Hoare). Choose an element as the **pivot**.

Partition the array so that all elements smaller than the pivot are on its left and all larger ones are on its right; the pivot is now in its final position. Recursively sort the left and right parts.

Partition (Lomuto scheme, pivot = last element): $i = \text{low} - 1$; for $j = \text{low}$ to $\text{high} - 1$: if $a[j] < \text{pivot}$, $i++$ and swap $a[i]$, $a[j]$. Finally swap $a[i + 1]$ with $a[\text{high}]$; the pivot is at index $i + 1$.

Worked example: sort {10, 80, 30, 90, 40, 50, 70} with pivot = last element (70).

- Elements less than 70: 10, 30, 40, 50 are moved to the front: 10 30 40 50 | 80 90 | 70 → swap pivot into place: **10 30 40 50 70 90 80**. 70 is now at index 4.
- Left part {10, 30, 40, 50}: pivot 50 → already in place; recursively {10, 30, 40} with pivot 40, etc.
- Right part {90, 80}: pivot 80 → 80 90.
- Result: 10 30 40 50 70 80 90.

Complexity. Average and best case $O(n \log n)$ (the pivot splits the array roughly in half, giving $\log n$ levels of $O(n)$ work). Worst case $O(n^2)$ when the pivot is always the smallest or largest (e.g. already sorted data with last-element pivot); choosing a random or median-of-three pivot avoids this. In-place, not stable. In practice one of the fastest sorting methods.

```
#include <stdio.h>
void swap(int *x, int *y) { int t = *x; *x = *y; *y = t; }
int partition(int a[], int low, int high)
{
    int pivot = a[high], i = low - 1, j;
    for (j = low; j < high; j++)
        if (a[j] < pivot) swap(&a[++i], &a[j]);
    swap(&a[i + 1], &a[high]);
    return i + 1;
}
void quickSort(int a[], int low, int high)
{
    if (low < high) {
        int p = partition(a, low, high);
        quickSort(a, low, p - 1);
        quickSort(a, p + 1, high);
    }
}
int main(void)
{
    int a[] = {10, 80, 30, 90, 40, 50, 70}, n = 7, i;
    quickSort(a, 0, n - 1);
    for (i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
    return 0;
}
```

Output: 10 30 40 50 70 80 90.

Algorithm	Best	Average	Worst	Extra space	Stable?
Bubble sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$ stack	No
Selection sort (awareness)	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No
Insertion sort (awareness)	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Merge sort (awareness)	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes

5.9 Common mistakes to avoid in Module V

- Applying binary search to unsorted data.
- Writing $\text{high} = \text{mid}$ or $\text{low} = \text{mid}$ (instead of $\text{mid} - 1$ / $\text{mid} + 1$), which can cause an infinite loop.
- Forgetting that the inner loop of bubble sort must stop at $n - \text{pass} - 1$ to avoid $a[j + 1]$ going out of bounds.
- Forgetting the base case $\text{low} < \text{high}$ in quick sort.
- Mixing up the order of traversals (remember: pre = root first, in = root in the middle, post = root last).

5.10 Solved problems: Module V

Problem 1. Construct a BST by inserting 45, 15, 79, 90, 10, 55, 12, 20, 50 and give its inorder, preorder and postorder traversals.



45 root; 15 left; 79 right; 90 right of 79; 10 left of 15; 55 left of 79; 12 right of 10; 20 right of 15; 50 left of 55.

Inorder (sorted): 10 12 15 20 45 50 55 79 90.

Preorder: 45 15 10 12 20 79 55 50 90.

Postorder: 12 10 20 15 50 55 90 79 45.

Answer: As above; height = 3 (edges on the path 45 → 79 → 55 → 50 or 45 → 15 → 10 → 12).

Problem 2. Delete 15 from the BST of the previous problem.

15 has two children (10 and 20). Inorder successor = smallest in its right subtree = 20. Replace 15 by 20 and delete the old 20 (a leaf).

Answer: New inorder: 10 12 20 45 50 55 79 90.

Problem 3. Construct the binary tree whose inorder is D B E A F C G and preorder is A B D E C F G. Find its postorder.

Preorder's first element A is the root. In inorder, left of A is D B E (left subtree), right is F C G.

Left subtree preorder B D E: root B, inorder D B E ⇒ D left, E right. Right subtree preorder C F G: root C, F left, G right.

Answer: Postorder: D E B F G C A.

Problem 4. Draw (describe) the expression tree for $(a + b) * (c - d / e)$ and give its prefix and postfix forms.

Root $*$; left subtree $+$ with a, b ; right subtree $-$ with left c and right $/ (d, e)$.

Preorder (prefix): $* + a b - c / d e$. Postorder (postfix): $a b + c d e / - *$.

Answer: As above.

Problem 5. How many comparisons does binary search need in the worst case for 1000 elements? For linear search?

Binary: $\lceil \log_2(1000 + 1) \rceil = 10$. Linear: 1000.

Answer: 10 and 1000.

Problem 6. Trace binary search for key 72 in {2, 5, 8, 12, 16, 23, 38, 56, 72, 91}.

low 0, high 9, mid 4 (16) → low 5. mid 7 (56) → low 8. mid 8 (72) → found.

Answer: Found at index 8 after 3 comparisons.

Problem 7. Show all passes of bubble sort on {64, 34, 25, 12, 22}.

Pass 1: 34 25 12 22 64. Pass 2: 25 12 22 34 64. Pass 3: 12 22 25 34 64. Pass 4: 12 22 25 34 64 (no swaps).

Answer: Sorted 12 22 25 34 64; total comparisons (without early exit) = $4 + 3 + 2 + 1 = 10$.



Problem 8. Show the first partition of quick sort on {35, 50, 15, 25, 80, 20, 90, 45} with the last element (45) as pivot.

Elements < 45: 35, 15, 25, 20 move to the front in order. After the loop: 35 15 25 20 80 50 90 45 (i = 3). Swap a[4] (80) with pivot: 35 15 25 20 45 50 90 80.

Answer: Pivot 45 is at index 4; left part {35, 15, 25, 20}, right part {50, 90, 80}.

Problem 9. What is the maximum number of nodes in a binary tree of height 4 (root at level 0)? What is the minimum height of a binary tree with 20 nodes?

$2^5 - 1 = 31$. Minimum height = $\lceil \log_2 21 \rceil - 1 = 5 - 1 = 4$.

Answer: 31 nodes; height 4.

Problem 10. Write a program to count leaf nodes and total nodes in a binary tree.

```
#include <stdio.h>
#include <stdlib.h>
struct tnode { int data; struct tnode *left, *right; };
struct tnode *mk(int x, struct tnode *l, struct tnode *r)
{ struct tnode *t = malloc(sizeof *t); t->data = x; t->left = l; t->right = r; return t; }
int nodes(struct tnode *r) { return r ? 1 + nodes(r->left) + nodes(r->right) : 0; }
int leaves(struct tnode *r) { if (!r) return 0; if (!r->left && !r->right) return 1; return leaves(r->left) + leaves(r->right); }
int main(void)
{
    struct tnode *root = mk(1, mk(2, mk(4, NULL, NULL), mk(5, NULL, NULL)), mk(3, NULL, mk(6, NULL, NULL)));
    printf("Nodes = %d, Leaves = %d\n", nodes(root), leaves(root));
    return 0;
}
```

ANS: Nodes = 6, Leaves = 3 (4, 5, 6).

5.11 Practice problems with solutions: Module V

Q1. Give the preorder, inorder and postorder of the tree: root 1; 1's children 2 and 3; 2's children 4 and 5; 3's children 6 and 7.

Preorder N L R; inorder L N R; postorder L R N.

Answer: Pre: 1 2 4 5 3 6 7; In: 4 2 5 1 6 3 7; Post: 4 5 2 6 7 3 1.

Q2. Insert 8, 3, 10, 1, 6, 14, 4, 7, 13 into a BST and give the inorder traversal.

Inorder of any BST is sorted.

Answer: 1 3 4 6 7 8 10 13 14.

Q3. Which traversal of a BST gives the elements in descending order?

Reverse inorder: right, root, left.

Answer: R N L (reverse inorder).

Q4. In a full binary tree with 10 internal nodes, how many leaves are there?

Leaves = internal + 1.



Answer: 11.

Q5. Sort {7, 2, 9, 4} by bubble sort showing each pass.

Pass 1: 2 7 4 9. Pass 2: 2 4 7 9. Pass 3: no swaps.

Answer: 2 4 7 9.

Q6. Why is quick sort's worst case $O(n^2)$, and when does it occur?

If the pivot is always the smallest or largest element, one partition is empty and the other has $n - 1$ elements, giving n levels of recursion with $O(n)$ work each. With the last element as pivot, this happens for already sorted or reverse-sorted input.

Answer: Unbalanced partitions (e.g. sorted input with last-element pivot).

5.12 MCQs: Module V

1. A tree with n nodes has how many edges?

- (a) $2n$ (b) n (c) $n + 1$ (d) $n - 1$

☐ (d) — Connected, acyclic.

2. A node with no children is called

- (a) root (b) leaf (c) internal node (d) parent

☐ (b) — Terminal node.

3. In a binary tree, each node has at most

- (a) any number (b) 1 child (c) 2 children (d) 3 children

☐ (c) — Definition.

4. The maximum number of nodes at level 3 of a binary tree (root at level 0) is

- (a) 3 (b) 16 (c) 8 (d) 6

☐ (c) — 2^3 .

5. The maximum number of nodes in a binary tree of height 3 is

- (a) 15 (b) 8 (c) 7 (d) 16

☐ (a) — $2^4 - 1$.

6. Inorder traversal visits nodes in the order

- (a) L R N (b) N L R (c) R N L (d) L N R

☐ (d) — Root in the middle.

7. Preorder traversal visits

- (a) L R N (b) L N R (c) level by level (d) N L R

☐ (d) — Root first.

8. Postorder traversal visits

- (a) L N R (b) R L N (c) L R N (d) N L R

☐ (c) — Root last.

9. Inorder traversal of a BST gives

- (a) descending order (b) sorted (ascending) order (c) level order (d) random order

☐ (b) — BST property.

10. Level-order traversal uses a

- (a) stack (b) queue (c) heap (d) array only

☐ (b) — Breadth-first.



- 11.** In a BST, the smallest element is found by
(a) checking the root (b) inorder only (c) going left repeatedly from the root (d) going right repeatedly
□ (c) — Leftmost node.
- 12.** When deleting a BST node with two children, it is replaced by its
(a) inorder successor or predecessor (b) parent (c) leftmost leaf of the whole tree (d) root
□ (a) — Keeps BST order.
- 13.** Search time in a balanced BST is
(a) $O(n^2)$ (b) $O(\log n)$ (c) $O(1)$ (d) $O(n)$
□ (b) — Height $\approx \log n$.
- 14.** Search time in a skewed BST is
(a) $O(n \log n)$ (b) $O(1)$ (c) $O(\log n)$ (d) $O(n)$
□ (d) — Behaves like a list.
- 15.** For a node at index i in an array representation (0-based), the left child is at
(a) $2i + 1$ (b) $i + 1$ (c) $i/2$ (d) $2i$
□ (a) — Right child $2i + 2$.
- 16.** Postorder of an expression tree gives the
(a) postfix (b) infix (c) prefix (d) none
□ (a) — Operands before operator.
- 17.** Binary search requires the data to be
(a) unsorted (b) in a linked list (c) sorted (d) of even size
□ (c) — Divide by comparison.
- 18.** Worst-case time of linear search is
(a) $O(\log n)$ (b) $O(n)$ (c) $O(n^2)$ (d) $O(1)$
□ (b) — Check every element.
- 19.** Worst-case time of binary search is
(a) $O(n \log n)$ (b) $O(n)$ (c) $O(1)$ (d) $O(\log n)$
□ (d) — Halves the range.
- 20.** For 1024 sorted elements, binary search needs at most about
(a) 512 comparisons (b) 1024 comparisons (c) 10 comparisons (d) 11 comparisons
□ (d) — $\lfloor \log_2 1024 \rfloor + 1 = 11$.
- 21.** Bubble sort compares
(a) adjacent elements (b) pivot with all (c) random pairs (d) the first and last elements
□ (a) — Adjacent swaps.
- 22.** The worst-case complexity of bubble sort is
(a) $O(n^2)$ (b) $O(n)$ (c) $O(n \log n)$ (d) $O(\log n)$
□ (a) — $n(n - 1)/2$ comparisons.
- 23.** The best case of bubble sort (with a swap flag) is
(a) $O(n^2)$ (b) $O(n)$ (c) $O(\log n)$ (d) $O(1)$
□ (b) — Already sorted.
- 24.** After the first pass of bubble sort (ascending),
(a) the array is sorted (b) nothing changes (c) the largest element is at the end (d) the smallest element is at the start
□ (c) — Bubbles up.



- 25.** Quick sort is based on
(a) divide and conquer (b) backtracking (c) greedy method (d) dynamic programming
□ (a) — Partition + recursion.
- 26.** The average-case complexity of quick sort is
(a) $O(n^2)$ (b) $O(n)$ (c) $O(\log n)$ (d) $O(n \log n)$
□ (d) — Balanced partitions.
- 27.** The worst case of quick sort occurs when
(a) array is random (b) n is small (c) pivot is always the smallest or largest element (d) pivot is the median
□ (c) — Unbalanced partitions.
- 28.** After partitioning in quick sort, the pivot is
(a) at index 0 (b) removed (c) in its final sorted position (d) at the end
□ (c) — Key property.
- 29.** Quick sort was developed by
(a) Hoare (b) Ritchie (c) Knuth (d) Dijkstra
□ (a) — C.A.R. Hoare, 1960.
- 30.** Which sorting algorithm is stable?
(a) bubble sort (b) both (c) quick sort (d) neither
□ (a) — Equal elements not swapped.
- 31.** The number of comparisons of bubble sort for 5 elements (no early exit) is
(a) 10 (b) 5 (c) 25 (d) 20
□ (a) — $4 + 3 + 2 + 1$.
- 32.** Which is a non-linear data structure?
(a) stack (b) queue (c) tree (d) array
□ (c) — Hierarchical.
- 33.** The height of a tree with only the root node is
(a) 2 (b) -1 (c) 1 (d) 0
□ (d) — Zero edges (edge convention).
- 34.** A binary tree in which every node has 0 or 2 children is
(a) empty (b) complete (c) full (strict) (d) skewed
□ (c) — Definition.
- 35.** Which pair of traversals uniquely defines a binary tree?
(a) inorder and preorder (b) only postorder (c) only inorder (d) preorder and postorder
□ (a) — Inorder + one other.
- 36.** In a BST, a new element is always inserted as
(a) the root (b) an internal node (c) a leaf (d) the leftmost node
□ (c) — At the NULL position.
- 37.** Which search works on a linked list efficiently?
(a) linear search (b) both equally (c) binary search (d) neither
□ (a) — No random access.
- 38.** $\text{mid} = \text{low} + (\text{high} - \text{low})/2$ is preferred to $(\text{low} + \text{high})/2$ because
(a) it avoids integer overflow (b) it gives a different result always (c) it is required by C (d) it is faster
□ (a) — Large indices.
- 39.** Which traversal is used to delete (free) all nodes of a tree safely?
(a) level order only (b) preorder (c) postorder (d) inorder
□ (c) — Children freed before parent.



40. A skewed binary tree with n nodes has height

(a) 1 (b) $n - 1$ (c) $\log n$ (d) $n/2$

□ (b) — Like a list.

Additional Solved Programs (All Modules)

Module I: patterns and number programs

Program A1: Pyramid star pattern (Lab 4.3).

```
#include <stdio.h>
int main(void)
{
    int n = 5, i, j;
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= n - i; j++) printf(" ");
        for (j = 1; j <= 2 * i - 1; j++) printf("*");
        printf("\n");
    }
    return 0;
}
```

Row i has $(n - i)$ spaces followed by $(2i - 1)$ stars, giving 1, 3, 5, 7, 9 stars.

Program A2: Floyd's triangle and number pyramid.

```
#include <stdio.h>
int main(void)
{
    int i, j, k = 1;
    for (i = 1; i <= 4; i++) {
        for (j = 1; j <= i; j++) printf("%3d", k++);
        printf("\n");
    }
    return 0;
}
```

Output: 1 / 2 3 / 4 5 6 / 7 8 9 10.

Program A3: Prime numbers between two given numbers (Lab 4.2).

```
#include <stdio.h>
int isPrime(int n)
{
    int i;
    if (n < 2) return 0;
    for (i = 2; i * i <= n; i++) if (n % i == 0) return 0;
    return 1;
}
int main(void)
{
    int a, b, n;
    printf("Enter range: ");
    scanf("%d %d", &a, &b);
    for (n = a; n <= b; n++) if (isPrime(n)) printf("%d ", n);
    printf("\n");
    return 0;
}
```

Input 10 30 → 11 13 17 19 23 29.

Program A4: Salary statement (basic, DA, HRA, TA) (Lab 2.3).



```
#include <stdio.h>
int main(void)
{
    float basic, da, hra, ta, gross;
    printf("Enter basic pay: ");
    scanf("%f", &basic);
    da = 0.50f * basic; hra = 0.20f * basic; ta = 0.10f * basic;
    gross = basic + da + hra + ta;
    printf("----- SALARY STATEMENT -----\n");
    printf("Basic Pay : %10.2f\n", basic);
    printf("DA (50%%) : %10.2f\n", da);
    printf("HRA (20%%) : %10.2f\n", hra);
    printf("TA (10%%) : %10.2f\n", ta);
    printf("Gross Pay : %10.2f\n", gross);
    return 0;
}
```

For basic 30000: DA 15000.00, HRA 6000.00, TA 3000.00, Gross 54000.00. (The DA/HRA/TA percentages are examples; use the values given in the question.)

Program A5: Student grade from marks in 4 subjects (Lab 3.1).

```
#include <stdio.h>
int main(void)
{
    int m[4], i, total = 0;
    float per;
    char grade;
    printf("Enter marks in 4 subjects: ");
    for (i = 0; i < 4; i++) { scanf("%d", &m[i]); total += m[i]; }
    per = total / 4.0f;
    if (per >= 90) grade = 'O';
    else if (per >= 80) grade = 'E';
    else if (per >= 70) grade = 'A';
    else if (per >= 60) grade = 'B';
    else if (per >= 50) grade = 'C';
    else if (per >= 40) grade = 'D';
    else grade = 'F';
    printf("Total = %d, Aggregate = %.2f%%, Grade = %c\n", total, per, grade);
    return 0;
}
```

Module II: arrays, strings and matrices

Program B1: Sort an array in ascending order and display unsorted and sorted arrays (Lab 5.2).

```
#include <stdio.h>
int main(void)
{
    int a[] = {29, 3, 72, 15, 8}, n = 5, i, j, t;
    printf("Unsorted: ");
    for (i = 0; i < n; i++) printf("%d ", a[i]);
    for (i = 0; i < n - 1; i++)
        for (j = i + 1; j < n; j++)
            if (a[i] > a[j]) { t = a[i]; a[i] = a[j]; a[j] = t; }
    printf("\nSorted: ");
    for (i = 0; i < n; i++) printf("%d ", a[i]);
    printf("\n");
    return 0;
}
```

(This is selection-exchange sort: each pass places the smallest remaining element at position i.)

Program B2: Matrix multiplication using functions (Lab 6.3).

```
#include <stdio.h>
#define N 10
void readM(int m[][N], int r, int c)
{
    int i, j;
    for (i = 0; i < r; i++) for (j = 0; j < c; j++) scanf("%d", &m[i][j]);
}
void multiply(int a[][N], int b[][N], int c[][N], int r1, int c1, int c2)
{
    int i, j, k;
    for (i = 0; i < r1; i++)
        for (j = 0; j < c2; j++)
            for (c[i][j] = 0, k = 0; k < c1; k++) c[i][j] += a[i][k] * b[k][j];
}
void printM(int m[][N], int r, int c)
{
    int i, j;
    for (i = 0; i < r; i++) { for (j = 0; j < c; j++) printf("%5d", m[i][j]); printf("\n"); }
}
int main(void)
{
    int a[N][N], b[N][N], c[N][N], r1, c1, r2, c2;
    printf("Rows and columns of A: "); scanf("%d %d", &r1, &c1);
    printf("Rows and columns of B: "); scanf("%d %d", &r2, &c2);
    if (c1 != r2) { printf("Multiplication not possible\n"); return 0; }
    readM(a, r1, c1); readM(b, r2, c2);
    multiply(a, b, c, r1, c1, c2);
    printM(c, r1, c2);
    return 0;
}
```

Program B3: Count the frequency of each character in a string.

```
#include <stdio.h>
int main(void)
{
    char s[] = "programming";
    int freq[26] = {0}, i;
    for (i = 0; s[i]; i++) if (s[i] >= 'a' && s[i] <= 'z') freq[s[i] - 'a']++;
    for (i = 0; i < 26; i++) if (freq[i]) printf("%c:%d ", 'a' + i, freq[i]);
    printf("\n");
    return 0;
}
```

Output: a:1 g:2 i:1 m:2 n:1 o:1 p:1 r:2.

Module IV: queue using a linked list

Program D1: Queue with a linked list.

```
#include <stdio.h>
#include <stdlib.h>
struct node { int data; struct node *next; };
struct node *front = NULL, *rear = NULL;
void enqueue(int x)
{
    struct node *t = malloc(sizeof *t);
    t->data = x; t->next = NULL;
    if (rear == NULL) front = rear = t;
    else { rear->next = t; rear = t; }
}
int dequeue(void)
{
    struct node *t = front; int x;
    if (!t) { printf("Queue empty\n"); return -1; }
    x = t->data; front = t->next;
    if (front == NULL) rear = NULL;
    free(t);
    return x;
}
int main(void)
{
    enqueue(5); enqueue(10); enqueue(15);
    printf("%d ", dequeue()); printf("%d ", dequeue());
    enqueue(20);
    printf("%d ", dequeue()); printf("%d\n", dequeue());
    return 0;
}
```

Output: 5 10 15 20.

Module V: other simple sorts (awareness)

Program E1: Insertion sort and selection sort.

```
#include <stdio.h>
void insertionSort(int a[], int n)
{
    int i, j, key;
    for (i = 1; i < n; i++) {
        key = a[i];
        for (j = i - 1; j >= 0 && a[j] > key; j--) a[j + 1] = a[j];
        a[j + 1] = key;
    }
}
void selectionSort(int a[], int n)
{
    int i, j, min, t;
    for (i = 0; i < n - 1; i++) {
        for (min = i, j = i + 1; j < n; j++) if (a[j] < a[min]) min = j;
        t = a[i]; a[i] = a[min]; a[min] = t;
    }
}
int main(void)
{
    int a[] = {12, 11, 13, 5, 6}, b[] = {64, 25, 12, 22, 11}, i;
    insertionSort(a, 5); selectionSort(b, 5);
    for (i = 0; i < 5; i++) printf("%d ", a[i]);
    printf("| ");
    for (i = 0; i < 5; i++) printf("%d ", b[i]);
    printf("\n");
    return 0;
}
```

Output: 5 6 11 12 13 | 11 12 22 25 64.

Module III and IV: more complete programs

Program C1: Doubly linked list (insert at end, delete a value, display forward and backward).

```
#include <stdio.h>
#include <stdlib.h>
struct dnode { int data; struct dnode *prev, *next; };
struct dnode *head = NULL, *tail = NULL;
void insertEnd(int x)
{
    struct dnode *t = malloc(sizeof *t);
    t->data = x; t->next = NULL; t->prev = tail;
    if (tail) tail->next = t; else head = t;
    tail = t;
}
void deleteValue(int x)
{
    struct dnode *p = head;
    while (p && p->data != x) p = p->next;
    if (!p) return;
    if (p->prev) p->prev->next = p->next; else head = p->next;
    if (p->next) p->next->prev = p->prev; else tail = p->prev;
    free(p);
}
int main(void)
{
    struct dnode *p;
    insertEnd(1); insertEnd(2); insertEnd(3); insertEnd(4);
    deleteValue(3);
    printf("Forward: "); for (p = head; p; p = p->next) printf("%d ", p->data);
    printf("\nBackward: "); for (p = tail; p; p = p->prev) printf("%d ", p->data);
    printf("\n");
    return 0;
}
```

Output: Forward: 1 2 4 / Backward: 4 2 1.

Program C2: Reverse a string using a stack.

```
#include <stdio.h>
#include <string.h>
int main(void)
{
    char s[] = "STACK", st[20];
    int top = -1, i, n = (int)strlen(s);
    for (i = 0; i < n; i++) st[++top] = s[i];    /* push all */
    for (i = 0; i < n; i++) s[i] = st[top--];    /* pop all */
    printf("%s\n", s);
    return 0;
}
```

Output: KCATS. (The last character pushed is the first popped: LIFO.)

Program C3: Polynomial addition using linked lists. A polynomial such as $5x^3 + 4x^2 + 2$ is stored as nodes (coefficient, exponent) in decreasing order of exponent.

```
#include <stdio.h>
#include <stdlib.h>
struct term { int coef, exp; struct term *next; };
struct term *add(struct term **tail, struct term *head, int c, int e)
{
    struct term *t = malloc(sizeof *t);
    t->coef = c; t->exp = e; t->next = NULL;
    if (*tail) (*tail)->next = t; else head = t;
    *tail = t;
    return head;
}
struct term *polyAdd(struct term *a, struct term *b)
{
    struct term *h = NULL, *tl = NULL;
    while (a && b) {
        if (a->exp > b->exp) { h = add(&tl, h, a->coef, a->exp); a = a->next; }
        else if (a->exp < b->exp) { h = add(&tl, h, b->coef, b->exp); b = b->next; }
        else { h = add(&tl, h, a->coef + b->coef, a->exp); a = a->next; b = b->next; }
    }
    for (; a; a = a->next) h = add(&tl, h, a->coef, a->exp);
    for (; b; b = b->next) h = add(&tl, h, b->coef, b->exp);
    return h;
}
int main(void)
{
    struct term *p = NULL, *pt = NULL, *q = NULL, *qt = NULL, *r;
    p = add(&pt, p, 5, 3); p = add(&pt, p, 4, 2); p = add(&pt, p, 2, 0); /* 5x^3+4x^2+2 */
    q = add(&qt, q, 5, 1); q = add(&qt, q, 5, 0); /* 5x+5 */
    for (r = polyAdd(p, q); r; r = r->next)
        printf("%dx^%ds", r->coef, r->exp, r->next ? " + " : "\n");
    return 0;
}
```

Output: $5x^3 + 4x^2 + 5x^1 + 7x^0$.

Program C4: Menu-driven stack using a switch (typical lab/viva program).

```
#include <stdio.h>
#define MAX 10
int main(void)
{
    int st[MAX], top = -1, ch, x;
    int input[] = {1, 5, 1, 7, 3, 2, 3, 4}, k = 0; /* simulated choices */
    do {
        ch = input[k++];
        switch (ch) {
            case 1: x = input[k++];
                    if (top == MAX - 1) printf("Overflow\n");
                    else { st[++top] = x; printf("Pushed %d\n", x); }
                    break;
            case 2: if (top == -1) printf("Underflow\n");
                    else printf("Popped %d\n", st[top--]);
                    break;
            case 3: printf("Top = %d\n", top == -1 ? -1 : st[top]); break;
            case 4: printf("Exit\n"); break;
        }
    } while (ch != 4);
    return 0;
}
```



Output: Pushed 5 / Pushed 7 / Top = 7 / Popped 7 / Top = 5 / Exit. (In the lab, replace the input array with scanf so that the user chooses the operations.)

Find the Error (Debugging Exercises)

Q1. `int a[5]; for (i = 1; i <= 5; i++) scanf("%d", &a[i]);`

Valid indices are 0 to 4. The loop skips a[0] and writes a[5], which is out of bounds.

Answer: Use for (i = 0; i < 5; i++).

Q2. `char name[20]; scanf("%s", &name); if (name == "Ravi") printf("Hi");`

& is not needed for a string (harmless but incorrect style). Comparing strings with == compares addresses.

Answer: Use `scanf("%19s", name);` and `if (strcmp(name, "Ravi") == 0)`.

Q3. `int *p; *p = 10;`

p is uninitialised (wild pointer); writing through it is undefined behaviour.

Answer: Initialise p: `int x; int *p = &x;` or `p = malloc(sizeof *p);`.

Q4. `if (x = 5) printf("five");`

Assignment instead of comparison; the condition is always true (5 is non-zero).

Answer: Use `if (x == 5)`.

Q5. `for (i = 0; i < 10; i++); { sum += i; }`

The semicolon after for ends the loop; the block runs only once, after the loop.

Answer: Remove the semicolon.

Q6. `float avg = sum / n;` (sum and n are int)

Integer division truncates before conversion to float.

Answer: Use `avg = (float)sum / n;`

Q7. `int f(int n) { return n * f(n - 1); }`

No base case: infinite recursion leading to stack overflow.

Answer: Add `if (n <= 1) return 1;`

Q8. `while (i < 10) printf("%d", i);`

i is never changed inside the loop: infinite loop.

Answer: Add `i++` inside the loop body.

Q9. `switch (grade) { case 'A': printf("Excellent"); case 'B': printf("Good"); }`

No break after case 'A', so it falls through and prints both messages.

Answer: Add break; after each case.

Q10. `int *p = malloc(10 * sizeof(int)); ... free(p); printf("%d", p[0]);`

Using memory after it has been freed (dangling pointer).

Answer: Do not access p after free; set `p = NULL` after freeing.



Q11. `printf("%d", 3.5);`

The format specifier does not match the argument type (double): undefined behaviour, prints garbage.

Answer: Use `%f` (or `%.1f`).

Q12. `char s[5] = "Hello";`

"Hello" needs 6 bytes including '\0'; the array is too small, so there is no terminator.

Answer: Use `char s[6] = "Hello";` or `char s[] = "Hello";`.



Short-Answer Questions (2 marks each) with Answers

Module I

Q1. What is an algorithm? List its characteristics.

A finite, ordered set of unambiguous steps to solve a problem. Characteristics: input, output, definiteness, finiteness, effectiveness.

Q2. Why is C called a middle-level language?

It supports high-level constructs (functions, data types, structured programming) and low-level operations (pointers, bitwise operators, direct memory access).

Q3. What is a token? Name the types of tokens in C.

The smallest meaningful unit of a program. Types: keywords, identifiers, constants, strings, operators, special symbols.

Q4. Differentiate between a keyword and an identifier.

A keyword is a reserved word with a fixed meaning (int, while); an identifier is a user-defined name for a variable or function (total, area).

Q5. What is the difference between = and ==?

= is the assignment operator (a = 5 stores 5 in a); == is the relational equality operator (a == 5 tests whether a equals 5, giving 1 or 0).

Q6. What is the difference between ++i and i++?

++i increments i before its value is used (prefix); i++ uses the current value and then increments (postfix).

Q7. What is type casting? Give an example.

Explicit conversion of a value from one type to another: (float)7/2 gives 3.5 instead of 3.

Q8. Distinguish between while and do-while loops.

while tests the condition before the body (may execute zero times); do-while tests after the body (executes at least once).

Q9. Differentiate break and continue.

break terminates the loop (or switch) immediately; continue skips the rest of the current iteration and goes to the next iteration.

Q10. What is the purpose of the default case in switch?

It executes when none of the case labels matches the switch expression. It is optional.

Q11. What are escape sequences? Give two examples.



Character combinations beginning with backslash that represent special characters: \n (newline), \t (tab).

Q12. What is the use of the sizeof operator?

It returns the size in bytes of a data type or variable, e.g. `sizeof(int) = 4` on typical systems.

Module II

Q1. What is a function prototype? Why is it needed?

A declaration giving the function's name, return type and parameter types, e.g. `int sum(int, int);`. It lets the compiler check calls before the definition appears.

Q2. What is recursion? What is a base case?

Recursion is a function calling itself. The base case is the condition under which the function returns without calling itself, stopping the recursion.

Q3. Differentiate between actual and formal parameters.

Actual parameters (arguments) are the values passed in the function call; formal parameters are the variables in the function definition that receive them.

Q4. What is an array? How are its elements stored?

A collection of elements of the same type under one name, stored in contiguous memory locations and accessed by index starting from 0.

Q5. How is a 2-D array stored in memory in C?

In row-major order: all elements of row 0, then row 1, and so on. Address of `a[i][j]` = `base + (i × cols + j) × size`.

Q6. What is a string in C? How does it differ from a character array?

A string is a character array terminated by `'\0'`. A character array need not contain `'\0'`; without it, it cannot be used with string functions.

Q7. Write the purpose of `strcmp()` and its return values.

Compares two strings lexicographically: returns 0 if equal, a negative value if the first is smaller, a positive value if it is larger.

Q8. Differentiate between a structure and a union.

In a structure each member has its own memory (size $\geq$ sum of members); in a union all members share the same memory (size = largest member), so only one member holds a valid value at a time.

Q9. What is typedef?



A keyword that creates an alias for an existing type, e.g. typedef struct student Student;.

Q10. How are arrays passed to functions?

By passing the address of the first element (the array name); changes made inside the function affect the original array.

Module III

Q1. What is a pointer? How is it declared?

A variable that stores the address of another variable. Declared as type *name; e.g. int *p;.

Q2. What do & and * mean with pointers?

& gives the address of a variable; * (dereference) gives the value stored at the address held in a pointer.

Q3. What is a NULL pointer?

A pointer that points to nothing, assigned the value NULL. It is used to indicate "no address" and is checked before dereferencing.

Q4. What are a wild pointer and a dangling pointer?

A wild pointer is uninitialised (contains garbage). A dangling pointer points to memory that has been freed or to a local variable that no longer exists.

Q5. What is pointer arithmetic? Give an example.

Adding or subtracting integers to/from pointers, which moves them by multiples of the element size: if int *p = 1000, then $p + 1 = 1004$.

Q6. Differentiate between call by value and call by reference.

Call by value passes copies of values (changes don't affect the caller); call by reference passes addresses (changes through pointers affect the caller's variables).

Q7. Differentiate malloc() and calloc().

malloc(size) allocates one block without initialisation; calloc(n, size) allocates n blocks and initialises them to zero.

Q8. What is the use of realloc() and free()?

realloc changes the size of an allocated block; free releases dynamically allocated memory back to the heap.

Q9. What is a memory leak?

Dynamically allocated memory that is never freed and can no longer be accessed because its address has been lost.



Q10. What is a pointer to pointer?

A variable that stores the address of another pointer, e.g. `int **pp = &p;`.

Module IV

Q1. What is a data structure? Classify data structures.

A way of organising data for efficient use. Primitive (int, char) and non-primitive; non-primitive are linear (array, list, stack, queue) or non-linear (tree, graph).

Q2. What is a linked list? Give one advantage over arrays.

A linear collection of nodes, each containing data and a pointer to the next node. Its size can change at run time and insertion/deletion needs only pointer changes.

Q3. What is a self-referential structure?

A structure that contains a pointer to a structure of the same type, e.g. `struct node { int data; struct node *next; };`

Q4. Define stack and its basic operations.

A LIFO linear structure where insertion and deletion occur at the top. Operations: push, pop, peek, isEmpty, isFull.

Q5. What are stack overflow and underflow?

Overflow: pushing onto a full stack. Underflow: popping from an empty stack.

Q6. Why is postfix notation used for evaluating expressions?

It needs no parentheses or precedence rules and can be evaluated in one left-to-right scan using a stack.

Q7. Convert $A * B + C / D$ to postfix.

`A B * C D / +.`

Q8. What is a circular queue? Why is it needed?

A queue in which the last position is followed by the first (indices wrap using % MAX). It reuses empty slots at the front that a linear queue cannot.

Q9. Give two applications each of stacks and queues.

Stack: function calls/recursion, expression evaluation. Queue: CPU scheduling, printer spooling.

Module V

Q1. Define binary tree and binary search tree.



A binary tree is a tree where each node has at most two children. A BST is a binary tree where left-subtree values are smaller and right-subtree values are larger than the node.

Q2. What are the three depth-first traversals of a binary tree?

Preorder (root, left, right), inorder (left, root, right), postorder (left, right, root).

Q3. What is special about the inorder traversal of a BST?

It visits the keys in ascending sorted order.

Q4. What is the condition for applying binary search?

The array must be sorted (and allow random access).

Q5. Compare the time complexities of linear and binary search.

Linear search $O(n)$; binary search $O(\log n)$.

Q6. What is the principle of bubble sort?

Repeatedly compare adjacent elements and swap them if out of order; after each pass the largest remaining element moves to its final position.

Q7. What is a pivot in quick sort?

The element around which the array is partitioned: smaller elements go to its left, larger to its right, and the pivot ends in its final sorted position.

Q8. Give the worst-case and average-case complexity of quick sort.

Worst $O(n^2)$; average $O(n \log n)$.



MCQ Practice Set 2: Predict the Output

1. `printf("%d", 10 / 3 * 3);`
(a) 1 (b) 3 (c) 9 (d) 10
□ (c) — $(10/3) = 3; 3 \times 3$.
2. `printf("%d", 5 + 3 % 2);`
(a) 0 (b) 4 (c) 6 (d) 5
□ (c) — $3 \% 2 = 1$ first.
3. `int a = 3; printf("%d", a << 3);`
(a) 3 (b) 6 (c) 24 (d) 9
□ (c) — 3×2^3 .
4. `printf("%d", 7 & 3);`
(a) 0 (b) 4 (c) 3 (d) 7
□ (c) — $111 \& 011 = 011$.
5. `printf("%d", 6 ^ 3);`
(a) 5 (b) 3 (c) 9 (d) 7
□ (a) — $110 \wedge 011 = 101$.
6. `printf("%d", !5);`
(a) 0 (b) -5 (c) 1 (d) 5
□ (a) — Logical NOT of non-zero.
7. `printf("%d", 4 > 3 && 3 > 4);`
(a) 3 (b) 4 (c) 1 (d) 0
□ (d) — $1 \&\& 0$.
8. `printf("%d", 4 > 3 || 3 > 4);` (`||` = logical OR)
(a) error (b) 0 (c) 1 (d) 7
□ (c) — $1 \text{ OR } 0 = 1$.
9. `int x = 10; printf("%d", x++ + 5);` then x is
(a) 15 and 10 (b) 15 and 11 (c) 16 and 11 (d) 16 and 10
□ (b) — Postfix uses 10.
10. `int i = 5; i = i++ * 0 + ++i;` (result)
(a) always 6 (b) undefined behaviour (c) well defined 7 (d) compile error
□ (b) — Modified twice without sequence point.
11. `printf("%.3f", 22.0 / 7);`
(a) 3 (b) 3.14 (c) 3.142 (d) 3.143
□ (d) — 3.142857... rounds to 3.143.
12. `printf("%c", 'a' - 32);`
(a) 65 (b) error (c) a (d) A
□ (d) — ASCII 97 - 32 = 65.
13. `printf("%d", 'B' - 'A');`
(a) B (b) 1 (c) 66 (d) 65
□ (b) — 66 - 65.
14. `int i; for (i = 0; i < 3; i++) printf("%d", i);`
(a) 012 (b) 123 (c) 0123 (d) 12
□ (a) — $i = 0, 1, 2$.
15. `int i = 10; while (i > 0) i -= 3; printf("%d", i);`
(a) -1 (b) -2 (c) 1 (d) 0
□ (b) — $10 \rightarrow 7 \rightarrow 4 \rightarrow 1 \rightarrow -2$.

- 16.** `int i = 0; do { i++; } while (i < 0); printf("%d", i);`
(a) 1 (b) infinite loop (c) 0 (d) -1
□ (a) — Body runs once.
- 17.** `int s = 0, i; for (i = 1; i <= 10; i += 3) s += i; printf("%d", s);`
(a) 12 (b) 15 (c) 22 (d) 55
□ (c) — $1 + 4 + 7 + 10$.
- 18.** `int x = 2; switch (x) { case 1: printf("A"); case 2: printf("B"); case 3: printf("C"); break; default: printf("D"); }`
(a) BC (b) C (c) BCD (d) B
□ (a) — Falls through to case 3, then break.
- 19.** `int a[] = {1, 2, 3, 4, 5}; printf("%d", a[a[1]]);`
(a) 4 (b) 3 (c) 2 (d) 1
□ (b) — $a[1] = 2 \rightarrow a[2] = 3$.
- 20.** `int a[3] = {5}; printf("%d%d", a[1], a[2]);`
(a) 00 (b) 55 (c) 50 (d) garbage
□ (a) — Rest are zero.
- 21.** `char s[] = "hello"; printf("%c", s[strlen(s) - 1]);`
(a) h (b) o (c) \0 (d) l
□ (b) — Last character.
- 22.** `char s[] = "abc"; printf("%zu", sizeof(s));`
(a) 8 (b) 3 (c) 4 (d) 1
□ (c) — Includes '\0'.
- 23.** `printf("%d", strcmp("B", "A") > 0);`
(a) 0 (b) -1 (c) 66 (d) 1
□ (d) — "B" > "A".
- 24.** `int f(int n) { return n <= 1 ? 1 : n * f(n - 1); } printf("%d", f(5));`
(a) 5 (b) 24 (c) 720 (d) 120
□ (d) — $5!$.
- 25.** `int g(int n) { return n == 0 ? 0 : 1 + g(n / 2); } printf("%d", g(8));`
(a) 4 (b) 8 (c) 3 (d) 2
□ (a) — $8 \rightarrow 4 \rightarrow 2 \rightarrow 1 \rightarrow 0$: 4 calls add 1.
- 26.** `int x = 5, *p = &x; *p += 3; printf("%d", x);`
(a) 5 (b) 3 (c) 8 (d) address
□ (c) — Changes x through p.
- 27.** `int a[] = {10, 20, 30}; int *p = a; p++; printf("%d", *p);`
(a) 10 (b) 30 (c) 20 (d) 11
□ (c) — Moves to $a[1]$.
- 28.** `int a[] = {10, 20, 30}; int *p = a; printf("%d", *p + 1);`
(a) 10 (b) 20 (c) 11 (d) 21
□ (c) — $(*p) + 1$.
- 29.** `int a[] = {10, 20, 30}; int *p = a; printf("%d", *(p + 1));`
(a) 10 (b) 20 (c) 11 (d) 30
□ (b) — $a[1]$.
- 30.** `int a[] = {1, 2, 3, 4}; int *p = a + 3; printf("%td", p - a);`
(a) 3 (b) 1 (c) 12 (d) 4
□ (a) — Element difference.



- 31.** `struct { int x; int y; } s = {3, 4}; printf("%d", s.x * s.y);`
(a) 0 (b) 34 (c) 7 (d) 12
□ (d) — 3×4 .
- 32.** `union { int i; char c; } u; u.i = 66; printf("%c", u.c);` (little-endian)
(a) garbage (b) A (c) 66 (d) B
□ (d) — Low byte = 66 = 'B'.
- 33.** `int x = 10; { int x = 20; printf("%d ", x); } printf("%d", x);`
(a) 20 10 (b) 20 20 (c) 10 20 (d) 10 10
□ (a) — Inner block shadows x.
- 34.** `void f(void) { static int c = 0; printf("%d", ++c); } f(); f(); f();`
(a) 000 (b) 123 (c) 012 (d) 111
□ (b) — Static retains value.
- 35.** `int top = -1, st[5]; st[++top] = 4; st[++top] = 9; printf("%d", st[top--] + st[top]);`
(a) 18 (b) undefined (top modified and read without sequence point) (c) well defined 13 (d) 8
□ (b) — Avoid such expressions; write in separate statements.
- 36.** Postfix `8 2 / 3 *` evaluates to
(a) 24 (b) 12 (c) 1 (d) 6
□ (b) — $8/2 = 4; 4 \times 3$.
- 37.** Stack after push 1, push 2, pop, push 3, push 4, pop; top element is
(a) 1 (b) 4 (c) 2 (d) 3
□ (d) — Stack: 1 3.
- 38.** Queue after enqueue 1, 2, 3, dequeue, enqueue 4; front element is
(a) 4 (b) 2 (c) 3 (d) 1
□ (b) — 1 removed.
- 39.** Inorder traversal of a BST built from 3, 1, 2 is
(a) 1 2 3 (b) 2 1 3 (c) 3 1 2 (d) 1 3 2
□ (a) — Sorted.
- 40.** Preorder of the BST built from 3, 1, 2 is
(a) 3 2 1 (b) 1 2 3 (c) 2 1 3 (d) 3 1 2
□ (d) — Root 3, left 1, 1's right 2.



Long-Answer Question Bank (with section references)

- Explain the structure of a C program with an example. Describe the compilation process (1.5).
- What is an algorithm? Write an algorithm and draw a flowchart to find the largest of three numbers (1.2, 1.3).
- Explain the data types of C with sizes, ranges and format specifiers (1.7).
- Explain the operators of C with examples; discuss precedence and associativity (1.10, 1.11).
- Explain decision-making statements (if, if-else, else-if ladder, switch) with examples (1.12).
- Explain the loops in C (for, while, do-while) with examples; compare while and do-while (1.13).
- What is a function? Explain the categories of functions, and call by value vs call by reference (2.1, 3.4).
- What is recursion? Write recursive programs for factorial, Fibonacci and GCD (2.2, 2.10).
- Explain 1-D and 2-D arrays. Write a program for matrix multiplication (2.3, 2.4).
- Explain strings and string-handling functions with examples (2.5).
- Differentiate between a structure and a union with examples (2.7, 2.8).
- What is a pointer? Explain pointer arithmetic and the relationship between pointers and arrays (3.1–3.3).
- Explain dynamic memory allocation functions with a program (3.7, 3.8).
- What is a linked list? Write functions for creation, insertion and deletion (4.2).
- Explain the stack with its array implementation. Explain infix-to-postfix conversion with an example (4.3–4.5).
- Write an algorithm to evaluate a postfix expression and trace it for an example (4.6).
- Explain linear and circular queues with programs (4.7).
- Explain binary trees and tree traversals with examples (5.1–5.3).
- What is a binary search tree? Explain insertion, search and deletion (5.4).
- Explain linear search and binary search with algorithms and complexity (5.5).
- Explain bubble sort and quick sort with examples; compare them (5.7, 5.8).



Model Question Paper with Solutions

Practice pattern · Time: 3 hours · Part A: 10 short questions (2 marks each) · Part B: long questions. Follow the exact pattern announced by your university.

Note: This is a practice paper written to match the syllabus. It is not an official university question paper.

Part A

Q1. What is the output of `printf("%d", 17 % 5 * 2 + 3);`?

$17 \% 5 = 2; 2 \times 2 = 4; 4 + 3 = 7.$

Q2. Which loop executes at least once?

do-while.

Q3. What is the size of union `{ int a; char b[10]; }` on a typical 32/64-bit system?

12 bytes (10 rounded up to a multiple of 4, the alignment of int).

Q4. What will `strlen("GSE\0BAM")` return?

3 (the string ends at the first '\0').

Q5. If `int a[] = {2, 4, 6, 8}; int *p = a + 1;`, what is `*(p + 2)`?

8.

Q6. Write the difference between `malloc` and `calloc` in one line.

`malloc` gives uninitialised memory; `calloc` gives zero-initialised memory for `n` elements.

Q7. Convert `(A + B) * (C - D)` to postfix.

`A B + C D - *.`

Q8. What is the condition for a circular queue to be full?

$(\text{rear} + 1) \% \text{MAX} == \text{front}.$

Q9. Give the inorder traversal of a BST built from 5, 3, 8, 1, 4.

1 3 4 5 8.

Q10. What is the worst-case complexity of bubble sort?

$O(n^2).$

Part B

Problem 1. Write a C program to check whether a given number is a palindrome.

```
#include <stdio.h>
int main(void)
{
    int n, t, rev = 0;
    printf("Enter a number: ");
    scanf("%d", &n);
    for (t = n; t > 0; t /= 10) rev = rev * 10 + t % 10;
    printf("%d is %sa palindrome\n", n, rev == n ? "" : "not ");
    return 0;
}
```

ANS: 12321 → palindrome; 12345 → not a palindrome.

Problem 2. Write a recursive function for factorial and trace fact(4).

$\text{fact}(4) = 4 \times \text{fact}(3) = 4 \times 3 \times \text{fact}(2) = 4 \times 3 \times 2 \times \text{fact}(1) = 4 \times 3 \times 2 \times 1 = 24$
(Section 2.2).

Answer: 24.

Problem 3. Write a program to swap two numbers using pointers and explain.

See Section 2.10 (swapByRef): the function receives addresses, so *a and *b refer to the caller's variables.

Answer: x = 20, y = 10 after swapping 10 and 20.

Problem 4. Write functions to insert at the beginning and delete from the end of a singly linked list.

```
#include <stdio.h>
#include <stdlib.h>
struct node { int data; struct node *next; };
void insertBegin(struct node **h, int x)
{ struct node *t = malloc(sizeof *t); t->data = x; t->next = *h; *h = t; }
void deleteEnd(struct node **h)
{
    struct node *p = *h;
    if (!p) return;
    if (!p->next) { free(p); *h = NULL; return; }
    while (p->next->next) p = p->next;
    free(p->next); p->next = NULL;
}
int main(void)
{
    struct node *h = NULL, *p;
    insertBegin(&h, 3); insertBegin(&h, 2); insertBegin(&h, 1);
    deleteEnd(&h);
    for (p = h; p; p = p->next) printf("%d ", p->data);
    printf("\n");
    return 0;
}
```

ANS: The list 1 2 3 becomes 1 2.



Problem 5. Convert $A + (B * C - (D / E ^ F) * G) * H$ to postfix using a stack.

Inner: $D / E ^ F \rightarrow D E F ^ /$; times $G \rightarrow D E F ^ / G *$; $B * C \rightarrow B C *$; subtract: $B C * D E F ^ / G * -$; times H ; add A .

Answer: $A B C * D E F ^ / G * - H * +$

Problem 6. Sort {33, 10, 55, 71, 29, 3, 18, 42} using quick sort (first partition with last element as pivot) and state the final result.

Pivot 42: elements < 42 (33, 10, 29, 3, 18) move left: 33 10 29 3 18 | 42 | 55 71 after placing the pivot. Recursively sorting both parts gives the result.

Answer: 3 10 18 29 33 42 55 71.

Problem 7. Construct a BST from 40, 25, 78, 10, 32, 50, 93, 3, 12, 35 and write all three traversals.

Inorder: 3 10 12 25 32 35 40 50 78 93.

Preorder: 40 25 10 3 12 32 35 78 50 93.

Postorder: 3 12 10 35 32 25 50 93 78 40.

Answer: As above.



Quick Reference Sheet (Last-Minute Revision)

Syntax summary

```
if (cond) { ... } else if (cond) { ... } else { ... }
switch (expr) { case 1: ...; break; default: ...; }
for (init; cond; update) { ... }
while (cond) { ... }
do { ... } while (cond);
type name(type param, ...) { ...; return value; } /* function */
type arr[N]; type mat[R][C]; char str[N] = "text";
struct tag { type m1; type m2; }; struct tag var, *p = &var; p->m1
union tag { type m1; type m2; };
type *p = &x; *p = value; p++; int **pp = &p;
p = malloc(n * sizeof *p); p = calloc(n, sizeof *p);
p = realloc(p, newn * sizeof *p); free(p); p = NULL;
```

Format specifiers

Specifier	Type	Specifier	Type
%d, %i	int	%u	unsigned int
%ld	long	%lld	long long
%f	float/double (printf)	%lf	double (scanf)
%c	char	%s	string
%x, %o	hexadecimal, octal	%e	scientific notation
%p	pointer address	%%	the % character
%zu	size_t (sizeof)	%5.2f	width 5, 2 decimals

Important library functions

Header	Functions
stdio.h	printf, scanf, getchar, putchar, fgets, puts, fopen, fclose
math.h	sqrt, pow, fabs, sin, cos, tan, log, exp, ceil, floor (link with -lm)
string.h	strlen, strcpy, strncpy, strcat, strcmp, strchr, strstr
stdlib.h	malloc, calloc, realloc, free, exit, abs, atoi, rand, qsort
ctype.h	isalpha, isdigit, isspace, isupper, islower, toupper, tolower

Data structure operations and complexities

Structure / algorithm	Key operations	Time
Array	access a[i]; insert/delete with shifting	O(1); O(n)
Linked list	insert/delete at head; search	O(1); O(n)
Stack (array or list)	push, pop, peek	O(1)
Queue (circular array or list)	enqueue, dequeue	O(1)



Structure / algorithm	Key operations	Time
BST (balanced / skewed)	search, insert, delete	$O(\log n)$ / $O(n)$
Linear search	—	$O(n)$
Binary search (sorted array)	—	$O(\log n)$
Bubble sort	—	$O(n^2)$ (best $O(n)$)
Quick sort	—	$O(n \log n)$ avg, $O(n^2)$ worst

Formulas and rules

Address of $a[i] = \text{base} + i \times \text{size}$; address of $a[i][j]$ (row-major) = $\text{base} + (i \times \text{cols} + j) \times \text{size}$

Pointer: $a[i] \equiv *(a + i)$; $p + k$ moves $k \times \text{sizeof}(*p)$ bytes

Stack (array): push $\rightarrow \text{stack}[++\text{top}] = x$; pop $\rightarrow \text{stack}[\text{top}--]$; empty: $\text{top} == -1$; full: $\text{top} == \text{MAX} - 1$

Circular queue: $\text{rear} = (\text{rear} + 1) \% \text{MAX}$; full: $(\text{rear} + 1) \% \text{MAX} == \text{front}$

Binary tree: max nodes at level $k = 2^k$; max nodes of height $h = 2^{h+1} - 1$; edges = $n - 1$

Binary search: $\text{mid} = \text{low} + (\text{high} - \text{low})/2$; at most $\lfloor \log_2 n \rfloor + 1$ comparisons

Bubble sort comparisons (worst) = $n(n - 1)/2$; Tower of Hanoi moves = $2^n - 1$

Traversal memory aid

Traversal	Order	BST result
Preorder	Root, Left, Right	root first (used to copy a tree)
Inorder	Left, Root, Right	sorted order
Postorder	Left, Right, Root	root last (used to delete a tree)



Glossary and ASCII Table

Term	Meaning
Algorithm	Finite sequence of unambiguous steps to solve a problem
Array	Collection of same-type elements in contiguous memory, accessed by index
Binary search tree	Binary tree where left subtree < node < right subtree
Call by reference	Passing addresses so a function can modify the caller's variables
Call by value	Passing copies of values; the caller's variables are unchanged
Compiler	Program that translates source code into machine (object) code
Dangling pointer	Pointer to memory that has been freed or no longer exists
Data structure	Organised way of storing data for efficient access and modification
Dynamic memory allocation	Requesting memory at run time from the heap (malloc, calloc, realloc)
FIFO	First In, First Out (queue)
Flowchart	Diagram of an algorithm using standard symbols
Function prototype	Declaration of a function's name, return type and parameter types
Heap	Memory region used for dynamic allocation
Identifier	User-defined name for a variable, function, array, etc.
Keyword	Reserved word with a predefined meaning in C
LIFO	Last In, First Out (stack)
Linked list	Chain of nodes, each holding data and a pointer to the next node
Linker	Combines object files and library code into an executable
Memory leak	Allocated memory that is never freed and whose address is lost
NULL	Special pointer value meaning "points to nothing"
Pivot	Element used to partition the array in quick sort
Pointer	Variable that stores the address of another variable
Postfix notation	Operator written after its operands (A B +)
Preprocessor	Handles #include and #define before compilation
Recursion	A function calling itself, with a base case to stop



Term	Meaning
Scope	Region of the program where a variable is visible
Stack (memory)	Memory region holding local variables and function-call records
Structure	User-defined type grouping members of different types
Token	Smallest unit of a program (keyword, identifier, constant, operator, ...)
Traversal	Visiting every element of a data structure once
Union	Like a structure, but all members share the same memory
Wild pointer	Uninitialised pointer containing a garbage address

Useful ASCII codes

Character	Code	Character	Code
'0' - '9'	48 - 57	space	32
'A' - 'Z'	65 - 90	'\n' (newline)	10
'a' - 'z'	97 - 122	'\t' (tab)	9
'\0' (null)	0	'#'	35
'+'	43	'.'	45
'*'	42	'/'	47

Useful conversions: digit character to number: $ch - '0'$; number to digit character: $n + '0'$; lower to upper case: $ch - 32$ (or $toupper(ch)$); upper to lower: $ch + 32$.

Viva voce quick questions

Q1. Who developed C and when?

Dennis Ritchie at AT&T Bell Laboratories in 1972.

Q2. What is the extension of a C source file and of an object file?

.c for source; .o (Linux) or .obj (Windows) for object files.

Q3. What does #include <stdio.h> do?

The preprocessor inserts the contents of the standard input/output header, which declares printf, scanf and related functions.

Q4. Why is main() declared as int main()?

main returns an integer status to the operating system: 0 for success, non-zero for failure.

Q5. What is the difference between a syntax error and a logical error?

A syntax error violates the language rules and is reported by the compiler; a logical error produces wrong output although the program compiles and runs.



Q6. What is the difference between a local and a global variable?

A local variable is declared inside a block and visible only there; a global variable is declared outside all functions and visible throughout the program.

Q7. What is an infinite loop? Give an example.

A loop whose condition never becomes false: `while (1) { ... }` or `for (;;) { ... }`.

Q8. Can a structure contain a pointer to itself?

Yes (self-referential structure), which is how linked lists and trees are built. It cannot contain an instance of itself.

Q9. What is the difference between stack memory and heap memory?

Stack memory holds local variables and is managed automatically; heap memory is allocated and freed explicitly with `malloc/free`.

Q10. Which data structure is used for recursion?

The (call) stack.



Exam Preparation Checklist

Programs you must be able to write from memory

Module	Program	Where in these notes
I	Largest of three numbers; even/odd; leap year	1.14, 1.16
I	Roots of a quadratic equation	Program 1.2
I	Sum of digits, reverse, palindrome number, Armstrong number	Program 1.3, 1.16, Model paper
I	Factorial, Fibonacci, prime check, primes in a range, GCD/LCM	Programs 1.4, 1.5, A3, 1.16
I	Simple and compound interest; salary statement; grade calculation	1.16, A4, A5
I	Pyramid and number patterns	A1, A2
II	Recursive factorial, Fibonacci, GCD, sum of digits, power, Tower of Hanoi	2.2, 2.10, 2.11
II	Array: max/min/sum/average, insertion, deletion, reverse, sort	2.10, B1
II	Matrix addition, transpose, multiplication (with functions)	2.4, 2.10, B2
II	String: palindrome, length/copy/concatenate without library, count vowels, word count	2.5, 2.10, 2.11
II	Structure: student records, employee salary; union size demonstration	2.7, 2.8, 2.10
III	Swap using pointers; sum and product via pointers	2.10, 3.4
III	Smallest element and position using pointer and function	3.10
III	malloc/calloc/realloc/free programs; dynamic 2-D array	3.8, 3.10
IV	Linked list: create, insert (begin/end/position), delete, display, count, search, reverse	4.2, 4.9
IV	Stack using array and linked list; parenthesis matching; string reversal	4.3, 4.9, C2
IV	Infix to postfix; postfix evaluation	4.5, 4.6
IV	Linear and circular queue; queue using linked list	4.7, D1



Module	Program	Where in these notes
V	BST insert, search, delete, traversals, height, count nodes and leaves	5.4, 5.10
V	Linear search, binary search (iterative and recursive)	5.5
V	Bubble sort, quick sort (and insertion/selection for awareness)	5.7, 5.8, E1

Theory topics to revise

- Characteristics of algorithms; flowchart symbols; structure of a C program; compilation steps.
- Data types with sizes and format specifiers; storage classes; type conversion.
- Operator precedence and associativity; prefix vs postfix increment; integer division and %.
- Differences: while vs do-while, break vs continue, if-else vs switch, recursion vs iteration, structure vs union, array vs linked list, stack vs queue, malloc vs calloc, call by value vs call by reference, static vs dynamic memory, linear vs binary search.
- Pointer arithmetic rules; array-pointer relationship; NULL, void, wild and dangling pointers.
- Infix/prefix/postfix conversion by hand with a stack table.
- Tree terms; three traversals by hand; BST deletion cases.
- Complexities of searching and sorting algorithms.

Tips for writing programs in the exam

- Always write #include lines and int main(void) with return 0;.
- Use meaningful variable names and indent the code properly.
- Declare all variables at the start of a block if your examiner expects C89 style.
- Show sample input and output after the program.
- For algorithms, write numbered steps; for data-structure operations, also show a small diagram or a trace table.
- Check boundary cases: empty list, one element, full stack, empty queue, $n = 0$ or 1 .