

UNIT I — Programming in C & Data Structures

1. Quick intro & history

- **C:** Developed by Dennis Ritchie (Bell Labs) early 1970s; portable, procedural language.
- **Why learn C:** foundation for systems programming, close to hardware, efficient, basis for C++/Java.
- **Program structure (typical):**

```
c Copy code  
  
#include <stdio.h>  
  
int main(void) {  
    // statements  
    return 0;  
}
```

2. Standard I/O in C

- Header: <stdio.h>
- Common functions:
 - printf, scanf — formatted text I/O.
 - getchar, putchar, gets(*unsafe, avoid*), fgets, puts.
 - File I/O: fopen, fclose, fread, fwrite, fprintf, fscanf.

Example:


```
c Copy code  
  
#include <stdio.h>  
  
int main(void) {  
    int n;  
    printf("Enter a number: ");  
    if (scanf("%d", &n) == 1)  
        printf("You entered %d\n", n);  
    return 0;  
}
```

3. C Declarations & Fundamental Data Types

- Basic types: char, int, short, long, float, double.
- Signed/unsigned variants: unsigned int.
- sizeof(type) gives storage size (bytes).
- void used for functions returning nothing or generic pointers (void *).

Example declaration:

```
c
int a;          // integer
unsigned int u; // non-negative integers
char c;         // single character
double x;       // double-precision float
```


 Copy code

4. Storage Classes

- **auto**: default for local variables (usually omitted).
- **register**: suggest storing in CPU register (compiler ignores as it likes).
- **static**:
 - For local: persists across function calls (retains value).
 - For global: internal linkage (visible only in the file).
- **extern**: declares variable defined in another file (external linkage).


Examples:

```
c
void f(void) {
    static int counter = 0; // persists between calls
    counter++;
    printf("%d\n", counter);
}
```

 Copy code

Global vs `static` global:

```
c
int x;          // global, external linkage
static int y;   // global, internal (file-scope)
```

 Copy code

5. Operators & Expressions

- Categories: arithmetic + - * / %, relational == != < > <= >=, logical && || !, bitwise & | ^ ~ << >>, assignment = +=, ternary ?:.
- **Operator precedence & associativity** matter — use parentheses to be explicit.
- **Mixed operands:** usual arithmetic conversions; smaller types promoted to int or double as needed.
- **Type conversion (casting):** (double) a / b.

Example (bit ops):

```
c Copy code  
  
unsigned int x = 5;      // 0101  
unsigned int y = x << 1; // 1010 = 10
```

6. Conditional Program Execution

- if, if-else, nested if, switch (integral types only).
- switch uses case labels; break to avoid fallthrough; default for else-case.

Example:

```
c Copy code  
  
int grade = 85;  
if (grade >= 75) printf("Distinction\n");  
else if (grade >= 60) printf("First Class\n");  
else printf("Keep trying\n");  
  
switch (grade / 10) {  
    case 10: case 9: printf("Excellent\n"); break;  
    case 8: printf("Very Good\n"); break;  
    default: printf("Other\n");  
}
```

7. Loops & Iteration

- while, do-while, for.
- break exits loop; continue skips to next iteration.
- Multiple loop variables allowed in for (comma operator).

Examples:

```
c
// for
for (int i = 0; i < n; ++i) { ... }

// while
int i = 0;
while (i < n) { ...; ++i; }

// do-while
int x;
do { x = getchar(); } while (x != '\n');
```

8. Modular Programming

- Break program into functions; use static for file-private helpers; use headers (.h) for declarations.
- **Pass by value:** C passes arguments by value (copies). To modify caller data, pass pointer.
- **Separate compilation & linkage:**
 - Put function declarations in header mylib.h.
 - Implement functions in mylib.c.
 - Compile: gcc -c mylib.c main.c then gcc -o prog main.o mylib.o.
- Use extern for shared global variables.

Example (two files):

mylib.h


```
c
#ifndef MYLIB_H
#define MYLIB_H
void greet(void);
#endif
```

mylib.c

```
c
#include <stdio.h>
#include "mylib.h"
void greet(void) { printf("Hello from lib\n"); }
```

main.c

c

 Copy code


```
#include "mylib.h"
int main(void){ greet(); return 0; }
```

9. Useful C concepts (pointers, arrays, strings — brief)

- **Pointers:** `int *p; p = &a; *p = 5;` Address arithmetic valid for arrays.
- **Arrays:** fixed-size; decay to pointer when passed to functions. Multi-dimensional arrays are row-major in memory.
- **Strings:** char arrays terminated by `'\0'`. Use `<string.h>` functions (`strlen`, `strcpy`, `strcmp`), but prefer `fgets` for safe reading.

Example pointer function:

c

 Copy code

```
void swap(int *a, int *b) {
    int t = *a; *a = *b; *b = t;
}
```

10. Common pitfalls & best practices

- **Buffer overflow:** avoid `gets()`. Use `fgets` with limits.
- **Uninitialized variables:** always initialize before use.
- **Dangling pointers:** after `free()`, set pointer to `NULL`.
- **Use `const`** where values shouldn't change.
- **Check return values** of `scanf`, `fopen`, `malloc`.
- **Format specifiers** must match types (`%d`, `%ld`, `%f`, `%lf`, `%c`, `%s`, `%zu`).

11. Small programs (exercise set)

1. **Hello + read int** — see earlier scanf example.
2. **Factorial (loop + recursion):**

```
c Copy code  
  
long fact_loop(int n) {  
    long f = 1;  
    for (int i=2; i<=n; ++i) f *= i;  
    return f;  
}  
long fact_rec(int n) {  
    if (n <= 1) return 1;  
    return n * fact_rec(n-1);  
}
```

3. Simple module: a `.h` / `.c` example shown above.

4. Bitwise parity:

```
c Copy code  
  
int parity(unsigned x) { return __builtin_parity(x); } // GCC builtin
```

12. Quick reference — operator precedence (most important)

Highest → `() [] -> .` → unary `! ~ ++ -- + - (type) * & sizeof` → `* / %` → `+ -` → `<< >>` → `< <= > >=` → `== !=` → `& ^ |` → `&& ||` → `?:` → `= += ...` Lowest.
(When in doubt, add parentheses.)

13. Exam tips (what to write)

- Always show **declarations** with types & memory class.
- For flow/loops, show initial values, loop condition, and update expression.
- For switch, mention case fall-through and use of break.
- For modular programming: explain header file usage, extern, and compilation commands.
- For storage classes: give one short example of behavior (e.g., static retains value).

14. Short checklist for revision

- Know printf/scanf, fopen/fread/fwrite.
 - Understand char/int/float/double, signed/unsigned.
 - Distinguish auto/register/static/extern.
 - Master operator types and precedence.
 - Be able to write if/switch, for/while/do constructs.
 - Practice splitting code into .c and .h and compiling.
-

15. Sample short questions (likely in exams)

1. Explain storage classes in C with examples.
2. Write a program to convert infix expression to postfix (Unit III will cover stacks; but practice basic versions).
3. Describe difference between malloc and calloc.
4. Show how to compile three source files into an executable using GCC.
5. Explain pointer arithmetic and how it relates to arrays.

UNIT II — Arrays, Structures, Pointers, Preprocessor & Standard Library

1. ARRAYS — notation, representation & operations

Definition & notation

- An **array** is a contiguous block of memory holding elements of same type.
- Declaration: type name[size]; e.g. int A[10];
- Indexing: elements accessed as A[i] where $0 \leq i < \text{size}$.

Memory layout (1-D)

- Layout: contiguous memory.
- Address calculation (important):
 - $\text{ADDR}(A[i]) = \text{BASE}(A) + i * \text{sizeof}(\text{element})$
 - $\text{sizeof}(\text{element})$ depends on type (e.g. $\text{sizeof}(\text{int})$).


Multidimensional arrays (row-major)

- `int M[r][c]`; stored row by row.
- Address of `M[i][j]`:
 - $\text{ADDR} = \text{BASE} + ((i * c) + j) * \text{sizeof}(\text{element})$
- For column-major languages the formula differs (C uses row-major).

Arrays of unknown / varying size

- Use dynamic allocation:


```
c
int n;
scanf("%d", &n);
int *A = malloc(n * sizeof *A); // or (int*)malloc(n * sizeof(int))
if (!A) { perror("malloc"); exit(1); }
...
free(A);
```

 Copy code

- Variable Length Arrays (VLAs) in C99: `int A[n];` (compiler dependent, not in C89).

Traversal (iterate)

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

 Copy code

Insertion & Deletion (array as sequential list)

- **Insertion at index k:** shift elements right from end to k $\rightarrow O(n)$ time.
- **Deletion at index k:** shift elements left from k+1 to end $\rightarrow O(n)$.

Example: insert

```
c Copy code

// assume A has capacity cap, current size n
void insert(int *A, int *n, int cap, int k, int val) {
    if (*n >= cap) return; // no space
    for (int i = *n; i > k; --i) A[i] = A[i-1];
    A[k] = val;
    (*n)++;
}
```

Sequential search (linear search)

- Walk array, compare each element; time complexity $O(n)$.

```
c Copy code

int linear_search(int *A, int n, int key) {
    for (int i=0; i<n; ++i) if (A[i]==key) return i;
    return -1;
}
```

Address calculation (useful for exams)

- For element size s and base address B:
 - $\text{Address}(A[i]) = B + i * s$
 - 2D: $\text{Address}(A[i][j]) = B + (i * \text{num_cols} + j) * s$

STRINGS

- In C, string = char array terminated by `'\0'`.
- Declaration: `char s[100];` or `char *s = "hello";`
- Standard functions in `<string.h>`: `strlen`, `strcpy`, `strncpy`, `strcat`, `strcmp`, `strchr`, `strstr`, `memcpy`, `memmove`.
- Safe input: `fgets(s, sizeof s, stdin);` (avoid `gets()`).

2. STRUCTURES (struct)

Purpose

- Group heterogeneous fields (like record/object).
- Example: represent student record.

Declaration & usage

```
c Copy code

struct Student {
    char name[50];
    int roll;
    float marks;
};

/* instantiate */
struct Student s1;
strcpy(s1.name, "Avi");
s1.roll = 101;
```

Assignment & passing

- In C, whole struct **assignment** is allowed (shallow copy of fields):

```
c Copy code

struct Student s2 = s1; // copies all fields
```

- Pass by value: copy. To avoid copying large structs pass pointer: `void f(struct Student *p)` and call `f(&s1)`.

Nested struct & arrays inside struct

```
c Copy code

struct Date { int d,m,y; };
struct Employee {
    char name[40];
    struct Date dob;
};
```

Memory layout & padding

- Compiler may insert **padding** for alignment — use `sizeof(struct ...)` to get actual bytes; order fields to reduce padding.

3. UNION

Purpose

- Save memory when variable holds one of many types at a time.
- All members share same memory location.

Syntax & example

```
c Copy code

union Value {
    int i;
    float f;
    char str[20];
};

/* usage */
union Value v;
v.i = 10;    // now v.f and v.str contain overlapping bytes
```

Important notes

- Only one member is valid at a time.
- Use with struct tags or separate enum to track active member (a discriminated union / tagged union).

4. POINTERS — addresses, arithmetic & pointers to objects

Basics

- Pointer stores address: `int *p; p = &x;`
- Dereference: `*p` gets/references value.

Pointer arithmetic

- `p + 1` advances by `sizeof(*p)` bytes (next element).
- For `int *p`, `p+1` points to next `int`.

Using pointers as function arguments

- Pass pointer to modify caller value:

```
c Copy code

void inc(int *p) { (*p)++; }
```

- For arrays: void process(int A[], int n) equivalently int *A.

Pointers & structs

- If struct S s; then pointer struct S *ps = &s; access via ps->field (equivalent to (*ps).field).


Dynamic memory allocation

- <stdlib.h> functions:
 - malloc(size): allocate uninitialized bytes. Returns void* (cast optional in C).
 - calloc(n, size): allocate and zero-initialize.
 - realloc(ptr, newsize): resize previously allocated block.
 - free(ptr): release memory.
- Always check return of malloc/calloc/realloc for NULL.
- Free memory once done to avoid leaks.

Examples

```
c
int *arr = malloc(n * sizeof *arr);
if (!arr) { perror("malloc"); exit(1); }

char *buf = calloc(100, sizeof *buf); // zeroed
arr = realloc(arr, new_n * sizeof *arr);
free(arr);
free(buf);
```

 Copy code

Common pointer pitfalls

- Dangling pointers after free.
- Double free.
- Using uninitialized pointers.
- Off-by-one indexing.

5. POINTERS TO FUNCTIONS (brief)

- Syntax: `int (*cmp)(const void*, const void*);`
 - Useful for callbacks (e.g., `qsort`).
 - Example: `qsort(array, n, sizeof(int), compare_ints);`
-

6. THE STANDARD C PREPROCESSOR

Directives

- `#include <stdio.h>` or `#include "my.h"`
- `#define NAME replacement (macros)`
- `#define SQR(x) ((x)*(x))` (use parentheses)
- `#undef NAME`
- Conditional compilation: `#ifdef DEBUG ... #endif`, `#ifndef`, `#if`, `#elif`, `#else`
- `#pragma compiler-specific`

Passing values to compiler

- Use `-DNAME` (e.g., `gcc -DDEBUG -o prog prog.c`) defines `DEBUG`.
- Example:

```
c Copy code  
  
#ifdef DEBUG  
    fprintf(stderr, "debug: x=%d\n", x);  
#endif
```

Good macro practices

- Use parentheses around macro params: `#define MAX(a,b) (((a)>(b))?(a):(b))`
- Prefer static inline functions for type-safety where possible (C99+).

7. THE STANDARD C LIBRARY (important I/O, strings, math)

Standard headers

- `<stdio.h>` — I/O, `fopen`, `fclose`, `fread`, `fwrite`, `fgets`, `fputs`, `fprintf`, `fscanf`
- `<stdlib.h>` — `malloc`, `calloc`, `realloc`, `free`, `qsort`, `bsearch`, `atoi`, `exit`
- `<string.h>` — string ops: `strlen`, `strcpy`, `strncpy`, `strcat`, `strcmp`, `memcpy`, `memmove`
- `<math.h>` — `sin`, `cos`, `tan`, `log`, `exp`, `pow`, `sqrt`
- `<errno.h>` — error codes & `perror`, `errno`

File I/O (basic)

```
c Copy code

FILE *fp = fopen("data.bin", "rb");
if (!fp) { perror("fopen"); exit(1); }

char buf[100];
while (fgets(buf, sizeof buf, fp) != NULL) {
    puts(buf);
}
fclose(fp);
```

Binary I/O

- `fwrite(&obj, sizeof obj, 1, fp);` and `fread(&obj, sizeof obj, 1, fp);`
- **For portability avoid writing raw structs with pointers — prefer explicit serialization.**


String handling

- `strlen(s)` — length (excluding `\0`)
- `strcpy(dest, src)` — copy (be careful buffer overflow)
- `strncpy` — copy up to `n` (no guarantee of `\0`)
- `strcmp` — compare
- `strchr`, `strstr` — search
- **`memcpy` vs `memmove` — `memmove` safe for overlapping regions.**

Math functions

- Use double or float variants. Link with `-lm` sometimes required.

c


 Copy code

```
double x = sin(3.1415 / 2.0);
double y = log(10.0);
```

8. EXERCISES & SAMPLE CODES (copy & test)

Linear search (array)


c

 Copy code

```
int linear_search(int *A, int n, int key) {
    for (int i = 0; i < n; ++i) if (A[i] == key) return i;
    return -1;
}
```

Dynamic array input (unknown size)


c

 Copy code

```
int *read_numbers(int *out_n) {
    int cap = 10, n = 0;
    int *a = malloc(cap * sizeof *a);
    int x;
    while (scanf("%d", &x) == 1) {
        if (n >= cap) {
            cap *= 2;
            int *tmp = realloc(a, cap * sizeof *tmp);
            if (!tmp) { free(a); return NULL; }
            a = tmp;
        }
        a[n++] = x;
    }
    *out_n = n;
    return a;
}
```


Struct + file write/read (text)

```
c
struct Person { char name[50]; int age; };
void save_person_txt(FILE *fp, const struct Person *p) {
    fprintf(fp, "%s\n%d\n", p->name, p->age);
}
```

 Copy code

Safe macro example

```
c
#define MAX(a,b) (((a) > (b)) ? (a) : (b))
```

 Copy code

9. COMPLEXITY & NOTES (for exams)

- Array traversal/search/insertion/deletion (shift) — $O(n)$ worst-case.
- Access by index — $O(1)$.
- Memory for array of n ints: $n * \text{sizeof}(\text{int})$.
- Pointer arithmetic uses `sizeof` implicitly to step elements.
- Use `calloc` when zero initialization required.

10. QUICK REVISION CHECKLIST

- Know array indexing & address formula (1D & 2D row-major).
- Be able to write insertion/deletion with shifts.
- Understand strings as `char[]` with `'\0'`.
- Know struct declaration, assignment, and pointer access `->`.
- Union shares memory — only one active member.
- Pointer arithmetic increments by element size.
- Know `malloc/calloc/realloc/free` & check for `NULL`.
- Macro syntax & conditional compilation (`#ifdef`, `-D`).
- Familiarity with `fopen/fread/fwrite/fclose`, `fgets/fputs`, `fprintf/fscanf`.
- `strlen/strcpy/strcmp/memcpy/memmove` usage and pitfalls.
- Math functions in `<math.h>`.

UNIT III — Data Structures (Complete, exam-ready notes)

1. Basic terminology & complexity

- **Data:** raw facts.
- **Data item / element:** fundamental unit (one value).
- **Data structure:** organization of data for efficient operations (array, list, tree, graph, etc.).
- **Abstract Data Type (ADT):** specification of operations and behavior (e.g., Stack ADT: push, pop, top, isEmpty).
- **Operation examples:** insert, delete, traverse, search, update.
- **Algorithm complexity:**
 - **Time complexity:** steps as function of n (big-O).
 - **Space complexity:** extra memory used (besides input).
- **Time-space tradeoff:** use extra memory (indexing, caches) to reduce time, or reduce memory and accept slower time.

2. STACKS

Definition & ADT

- **LIFO (Last In First Out). Operations:**
 - **push(x):** insert element on top.
 - **pop():** remove and return top element.
 - **top() / peek():** read top.
 - **isEmpty(), isFull().**

Representations

1. Array (static)

- **Use an array `stack[capacity]` and integer `top` (index of top element or -1 if empty).**
- **push: `stack[++top] = x` (check overflow).**
- **pop: `x = stack[top--]` (check underflow).**

2. Linked list (dynamic)

- Use singly linked nodes where head is top.
- **push: create new node** → **node->next = head; head = node.**
- **pop: remove head node.**

C-style array implementation (simplified)

```
c Copy code

#define CAP 100
int stack[CAP];
int top = -1;

int isEmpty(){ return top == -1; }
int isFull(){ return top == CAP-1; }

void push(int x){
    if(isFull()){ printf("Overflow\n"); return; }
    stack[++top] = x;
}

int pop(){
    if(isEmpty()){ printf("Underflow\n"); return INT_MIN; }
    return stack[top--];
}
```

Applications

- **Expression parsing: conversion infix \rightleftharpoons postfix/prefix.**
- **Evaluation of postfix expressions.**
- **Function call stack (recursion).**
- **Backtracking (DFS), undo mechanisms.**

Infix → Postfix (Shunting-yard idea, simple)

- **Use operator stack. Output list starts empty.**
- **Read token:**
 - **If operand → output.**
 - **If (→ push.**
 - **If) → pop operators to output until (.**
 - **If operator op → while top has operator of higher/equal precedence, pop to output; then push op.**
- **After input, pop remaining operators to output.**

Pseudocode (tokens):

```
css Copy code

for token in input:
    if operand: output.push(token)
    else if token == '(': opstack.push('(')
    else if token == ')': while(opstack.top()!='(') output.push(opstack.pop()); opstack.pop()
    else:
        while(opstack not empty and precedence(opstack.top) >= precedence(token))
            output.push(opstack.pop())
        opstack.push(token)
while(opstack not empty) output.push(opstack.pop())
```

Evaluate Postfix

- **Use stack of values.**
- **For token:**
 - **if operand: push numeric value**
 - **if operator: b = pop(); a = pop(); push(apply(op,a,b))**
- **Result = pop()**

Complexities: push/pop O(1). Conversion and evaluation O(n).

3. RECURSION

Definition


- **Function that calls itself. Base case + recursive case.**

Recursion vs Iteration

- **Recursion often clearer for divide & conquer, tree traversal, backtracking.**
- **Iteration can be more space/time efficient (recursion uses call stack).**

Example: Factorial


```
c
long fact(int n){
    if(n <= 1) return 1;
    return n * fact(n-1);
}
```

 Copy code

Tower of Hanoi (classical)

- **Move n disks from A to C using B:**

```
rust
Hanoi(n, A, B, C):
    if n==1: move A->C
    else:
        Hanoi(n-1, A, C, B)
        move A->C
        Hanoi(n-1, B, A, C)
```

 Copy code

Time complexity $T(n) = 2T(n-1) + 1 \rightarrow O(2^n)$.

Simulating recursion (stack)

- Recursive calls can be simulated using explicit stack for languages/environments without recursion.

Backtracking

- Use recursion to explore choices and backtrack on failure (e.g., n-queens, maze solve).

Tail recursion & removal

- **Tail recursion:** recursive call is the last action. Can be optimized to loops by compilers (tail-call optimization).
- To remove recursion, use iterative methods and an explicit stack where needed.

4. QUEUES

ADT

- FIFO (First In First Out). Operations:
 - enqueue(x) / add
 - dequeue() / remove
 - front() / peek
 - isEmpty, isFull

Representations

1. Array (simple)

- Use front and rear indices, but naive array needs shifting or uses circular indexing.

2. Circular queue (proper array)

- Use modulo arithmetic to wrap around:
 - $\text{rear} = (\text{rear} + 1) \% \text{capacity}; \text{queue}[\text{rear}] = x;$
 - $\text{front} = (\text{front} + 1) \% \text{capacity};$
 - Keep count or reserve one cell to distinguish full vs empty.

3. Linked list

- Maintain front and rear pointers; enqueue at rear, dequeue at front ($O(1)$).

Circular queue details

- Keep size variable or use condition $(\text{rear}+1)\% \text{cap} == \text{front}$ as full.
- Enqueue complexity $O(1)$, Dequeue $O(1)$.

Variants

- **Deque (double-ended queue):** insert/delete at both ends.
- **Priority queue:** elements have priority; support insert and delete-min (or delete-max). Implement with binary heap for $O(\log n)$ insert and extract.

Example: circular queue (simplified)

```
c Copy code  
  
int q[CAP];  
int front = 0, rear = 0; // empty when front==rear  
int isEmpty(){ return front==rear; }  
int isFull(){ return (rear+1)%CAP == front; }  
void enqueue(int x){ if(isFull()) error; q[rear]=x; rear=(rear+1)%CAP; }  
int dequeue(){ if(isEmpty()) error; int v=q[front]; front=(front+1)%CAP; return v; }
```

5. LINKED LISTS

Basic idea

- Nodes with data and next pointer. Dynamic size, efficient insert/delete ($O(1)$ when at head or known position).

Types

- **Singly linked list:** nodes have next.
- **Doubly linked list:** nodes have prev and next, supports backward traversal and $O(1)$ deletion given pointer.
- **Circular linked list:** last node links back to first, useful for round-robin.
- **Two-way header list:** header node contains metadata (length, first, last).

Node (C)

```
c Copy code  
  
typedef struct Node {  
    int data;  
    struct Node *next;  
} Node;
```

Basic operations (Singly)

- Insert at head:

```
c Copy code

Node* insert_head(Node* head, int x){
    Node* n = malloc(sizeof *n); n->data = x; n->next = head;
    return n; // new head
}
```

- Insert after a node p:

```
c Copy code

void insert_after(Node* p, int x){
    Node* n = malloc(...); n->data = x; n->next = p->next; p->next = n;
}
```

- Delete head:

```
c Copy code

Node* delete_head(Node* head){
    if(!head) return NULL;
    Node* t = head; head = head->next; free(t); return head;
}
```

- Delete after p:

```
c Copy code

void delete_after(Node* p){
    Node* t = p->next; if(!t) return; p->next = t->next; free(t);
}
```

Traversal & search

- Walk with pointer $p = \text{head}$; while(p){ if(p->data==key) break; p=p->next; } $O(n)$.

Insertion & deletion complexity

- Insert/delete at head: $O(1)$.
- Insert/delete at arbitrary position when node known: $O(1)$.
- Search to find position: $O(n)$.

Doubly linked list

- Node has prev and next. Easier deletion when node pointer known (no need to find predecessor).
- More memory overhead.

Linked List in Array (static linked list)

- Implement pointers as indices in array (useful in environments without dynamic allocation).

Polynomial representation & addition (linked list)

- Represent polynomial as linked list of terms sorted by exponent.
- Addition: traverse both lists, merge terms with same exponent.

Pseudo:

```
lua Copy code  
  
add(poly1, poly2):  
    result = empty  
    p = poly1; q = poly2  
    while p && q:  
        if p.exp == q.exp: sum = p.coeff+q.coeff; append term(sum, p.exp); p=p.next; q=q.next  
        else if p.exp > q.exp: append p; p=p.next  
        else append q; q=q.next  
    append remaining
```

Complexity $O(m+n)$.

Generalized linked list

- Node may hold data or pointer to sublist (tree-like). Useful for representing nested lists.

Garbage collection & compaction (brief)

- In languages with manual memory (C), programmer must free() unused nodes.
- Garbage collection (GC) automates reclaiming unreachable memory (mark-and-sweep, reference counting).
- Compaction (in GC) moves live objects to reduce fragmentation — not native in C; conceptually in managed languages.

6. Complexity summary (common operations)

Structure	Access by index	Search	Insert (known pos)	Delete (known pos)	Memory
Array (unsorted)	$O(1)$	$O(n)$	$O(n)$ (shift)	$O(n)$ (shift)	contiguous n
Stack (array/linked)	$O(n)$	$O(n)$	$O(1)$ push	$O(1)$ pop	$O(n)$
Queue (array/linked)	$O(n)$	$O(n)$	$O(1)$ enqueue	$O(1)$ dequeue	$O(n)$
Singly linked list	$O(n)$	$O(n)$	$O(1)$ (head)	$O(1)$ (head)	dynamic nodes
Doubly linked list	$O(n)$	$O(n)$	$O(1)$	$O(1)$	2 pointers/node

7. Exam tips & common pitfalls

- **For stack-based infix→postfix, carefully handle precedence and associativity; parentheses must be handled.**
- **When writing recursion, always include base case and prove termination.**
- **Off-by-one errors common in circular queue index updates; use modulo arithmetic and test small cases.**
- **Free nodes after deletion to avoid memory leak; set pointers to NULL after free to avoid dangling pointers.**
- **When merging sorted lists (e.g., polynomial addition), maintain order by exponent.**

8. Practice problems (recommended)

1. **Implement stack using linked list and use it to evaluate a postfix expression containing multi-digit numbers.**
2. **Write C function to convert an infix expression (with + - * / ^ and parentheses) to postfix.**
3. **Implement circular queue and simulate producer/consumer using it.**
4. **Implement singly linked list with functions to reverse list (iterative & recursive) and test.**
5. **Represent polynomials as linked lists and implement addition and multiplication.**
6. **Simulate recursion for Tower of Hanoi and count number of moves for n disks.**

UNIT IV— Trees, Graphs

TREES — basic terminology

- **Tree:** hierarchical ADT with nodes and parent-child links; one root, no cycles.
 - **Degree** of node = number of children.
 - **Leaf (external)** = node with degree 0. **Internal** = non-leaf.
 - **Height** of node = length (edges) of longest downward path; **height of tree** = height(root).
 - **Depth / level** = distance from root (root level = 0 or 1 depending).
-

Binary Trees

- Each node has ≤ 2 children: left and right.
- **Full (proper)** binary tree: every internal node has exactly two children.
- **Complete binary tree:** all levels full except possibly last, filled left→right.
- **Extended binary tree:** leaf/null children represented explicitly (useful for some proofs).

Representations


- **Array (implicit) representation:** good for complete trees: index 1..n where for node i: left= $2i$, right= $2i+1$, parent= $\lfloor i/2 \rfloor$.
 - **Linked representation:** nodes with left and right pointers (typical).
-

Binary tree traversals (recursive)

- **Inorder (L, Root, R)** — for BST yields sorted order.
- **Preorder (Root, L, R)** — useful for copying tree.
- **Postorder (L, R, Root)** — useful for deletion.

Pseudocode (inorder):

c

 Copy code

```
void inorder(node *r) {  
    if (r==NULL) return;  
    inorder(r->left);  
    visit(r);  
    inorder(r->right);  
}
```

Complexity: $O(n)$ visit each node once.

Threaded Binary Trees

- Null child pointers replaced with **threads** pointing to inorder predecessor/successor to allow non-recursive traversal and cheap successor access.
- **Single-threaded:** only one of left/right threads; **double-threaded:** both.
- Traversal uses threads to move to next node without stack or recursion.

Algebraic Expressions & Expression Trees

- Build expression tree where internal nodes are operators and leaves operands.
- Inorder traversal yields infix (with parentheses), preorder \rightarrow prefix, postorder \rightarrow postfix.

Huffman Algorithm (optimal prefix codes)

- Build a binary tree for variable-length prefix codes minimizing weighted path length:
 1. Start with forest of leaf nodes with weights (frequencies).
 2. Repeat: remove two smallest weight trees, create new node with weight = sum, insert back.
 3. Final tree gives codes by left/right bit assignments.
- Complexity: $O(n \log n)$ with priority queue.

Binary Search Trees (BST)

- BST property: left subtree keys < node.key < right subtree keys.
- **Search, insert, delete** average $O(h)$ where $h \approx \log n$ for balanced tree; worst $O(n)$ if degenerate.

BST Search (recursive)

```
c Copy code  
  
node* bst_search(node* r, int key) {  
    if(!r || r->key==key) return r;  
    return (key < r->key) ? bst_search(r->left, key) : bst_search(r->right, key);  
}
```

BST Insert (iterative)

- Walk down, attach as leaf maintaining BST property. $O(h)$.

BST Delete (cases)

1. Node is leaf \rightarrow remove directly.
2. Node has one child \rightarrow replace node with child.
3. Node has two children \rightarrow find inorder successor (min in right subtree) or predecessor, copy key, delete successor node (which will be case 1 or 2).

Balanced BSTs: AVL Trees

- **AVL tree**: binary search tree with balance factor $bf = \text{height}(\text{left}) - \text{height}(\text{right}) \in \{-1, 0, 1\}$ for every node.
- After insertion/deletion, restore balance using rotations:
 - Single rotations: LL (right rotation), RR (left rotation).
 - Double rotations: LR (left on left child then right), RL (right on right child then left).
- Search/insert/delete: $O(\log n)$.

Quick LL rotation (right rotate at x):



B-Trees (and B+ trees)

- Balanced m-ary search tree used in databases/filesystems.
- **Properties:**
 - Each node can have up to m children (order m).
 - Keys in internal nodes guide search; leaves at same depth.
 - B+ tree: all records in leaves; internal nodes only keys (good for range queries).
- **Use:** block-oriented storage (disk), minimize disk I/O.
- Operations: search $O(\log_m n)$ disk reads; insertion/deletion use splitting/merging and promote keys.

GRAPHS — terminology & representations

- **Graph $G = (V, E)$** ; **directed** (arcs) or **undirected**; multi-graph allows parallel edges; loops may be allowed.
- **Adjacency matrix:** $A[n][n]$ where $A[u][v] = 1$ or weight if edge exists. Space $O(n^2)$. Good for dense graphs and constant-time edge tests.
- **Adjacency list:** each vertex stores list of neighbors. Space $O(n + e)$. Good for sparse graphs.

Graph Traversal — BFS & DFS

- **Breadth-First Search (BFS):** uses queue; visits vertices by layers; computes shortest path distances (in unweighted graphs).
 - Complexity: $O(n + e)$.

- **Depth-First Search (DFS):** uses recursion/stack; useful for connectivity, topological sort (DAG), component discovery, backtracking and detecting cycles.
 - Complexity: $O(n + e)$.
 - **Connected components:** run DFS/BFS from unvisited vertices.
-

Spanning Trees & Minimum Spanning Tree (MST)

- **Spanning tree:** subset of edges connecting all vertices without cycles.
- **Minimum spanning tree:** spanning tree with minimum total weight.

Kruskal's algorithm (edge-based)

1. Sort edges by weight.
2. Use Union-Find (disjoint set) to add smallest edge that doesn't create cycle.
3. Complexity: $O(e \log e)$ (sorting) + union-find costs.

Prim's algorithm (vertex-based)

1. Start from a vertex, grow tree by repeatedly adding smallest weight edge connecting tree to outside vertex (use priority queue).
 2. Complexity $O(e \log n)$ with binary heap.
-

Shortest Paths (mention)

- **Dijkstra** for nonnegative weights (single-source) $O(e + n \log n)$ with heap.
- **Bellman-Ford** handles negative weights (detect negative cycles) $O(n e)$.

UNIT V — SEARCHING, HASHING, SORTING & FILE STRUCTURES

Sequential (Linear) search

- Scan array, compare each element. $O(n)$, simplest.

Binary search

- Requires sorted array. Repeatedly halve search interval. $O(\log n)$ time.
- Must handle boundaries carefully; iterative/recursive implementations.

Hashing (Hash table)

- **Hash function $h(\text{key})$** maps key domain \rightarrow table index $0..m-1$.
- **Load factor $\alpha = n / m$** (n = elements, m = table size).
- **Collision resolution strategies:**
 - **Chaining:** each slot has list of keys; insertion $O(1)$ avg, lookup $O(1 + \alpha)$.
 - **Open addressing (probing):** store element in table slots:
 - **Linear probing:** $h_i = (h + i) \% m$ — suffers clustering.
 - **Quadratic probing:** $h_i = (h + c_1 i + c_2 i^2) \% m$ — reduces clustering.
 - **Double hashing:** $h_i = (h + i * h_2(\text{key})) \% m$ — good distribution.
 - **Rehashing / dynamic resizing:** when α exceeds threshold (e.g., 0.7) allocate larger table and reinsert keys.
- Choice of hash function and load control critical for performance.

SORTING ALGORITHMS (overview & complexities)

Algorithm	Avg time	Worst time	Space	Stable?	Use case
Insertion sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	small n , nearly sorted
Bubble sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	educational
Selection sort	$O(n^2)$	$O(n^2)$	$O(1)$	No	small memory
Quick sort	$O(n \log n)$	$O(n^2)$ (bad pivot)	$O(\log n)$ avg	Not stable	general purpose (fast)
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	Stable	external sorting, stable requirement
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(1)$	Not stable	in-place guarantee
Two-way merge sort	variant for external merging				merges multiple runs

Quick Sort (outline)

- Choose pivot (random, median-of-3 recommended), partition array, recurse left & right.
- Average $O(n \log n)$, worst $O(n^2)$ for bad pivot choices.

Merge Sort (outline)

- Divide array in two halves, sort both halves, merge. Good for external sort (merge passes).

Heap Sort

- Build max-heap in $O(n)$, repeatedly extract max and heapify, yields in-place $O(n \log n)$

Practical considerations for internal sorting

- Memory constraints — choose in-place sorts.
- Stability: important when multiple keys/sorting by multiple fields (use mergesort or stable algorithms).
- Cache behavior: algorithms with sequential access (merge) may be more cache-friendly.

HASH TABLE IMPLEMENTATION (practical)

- Choose table size m (prefer prime or power-of-two depending on hash method).
- For chaining, store pointers to linked lists (separate chaining).
- For open addressing, ensure load factor < 0.7 typically.
- Handle deletion in open addressing with **tombstone markers** (special deleted marker) to preserve probe sequences.

FILE STRUCTURES (storage & indexing)

Physical storage basics

- **Block / disk sector / page:** unit of I/O; reading/writing processed in blocks.
- **Blocking factor** = number of records per block ($\text{floor}(\text{block_size} / \text{record_size})$).

File Organizations

- **Sequential files:** records stored in key order. Good for batch processing.
- **Heap (unordered) files:** append new records at end; fast inserts, slow searches.
- **Indexed files:** maintain index (like book index) mapping keys to record blocks for faster access.
- **Hash files:** use hashing to map key \rightarrow block(s).

Indexing

- **Primary index:** built on ordered primary key. Might be dense (entry per record) or sparse (entry per block).
- **Secondary index:** support queries on non-primary keys (may map to primary key or blocks).
- **B-Tree / B+ Tree index files:** balanced tree structures designed for disk blocks:
 - B+ tree: leaves contain data or pointers to data; internal nodes only keys. Efficient for range queries and sequential access.
 - B-tree: both internal and leaves contain records/keys.

B+ Tree vs B-Tree

- B+ tree stores all data pointers in leaves; internal nodes only guide search \rightarrow faster range scans, simpler leaf chaining for sequential access.
- Both are balanced; node size chosen to match disk block size to minimize I/O.

Indexing & Hashing comparison

- **Indexing (B-trees):** good for range queries, ordered traversal, can handle insert/delete with rebalancing.
- **Hashing:** good for point queries (exact key lookup). Not suitable for range queries. Performance sensitive to load factor and collisions.

Overflow handling (for hashed files)

- **Overflow chaining:** when bucket full, store extras in overflow area (disk chains).
 - **Open addressing** is less common for disk files due to need for in-block probing.
-

EXAM & IMPLEMENTATION TIPS

- For BST insert/delete, practice all cases (esp. deletion with two children). Draw small trees and simulate.
- For AVL, be able to show rotations for each imbalance case (LL, RR, LR, RL).
- For B-Tree, understand split on insert and merge/borrow on delete.
- For Huffman, practice building tree for small frequency sets.
- For graphs, practice BFS/DFS, Prim/Kruskal on sample graphs and track sets/priority queue/union-find steps.
- For hashing, illustrate chaining and linear probing examples showing collisions and probe sequences.
- For sorting, be able to explain quicksort partitioning and mergesort merging, and analyze worst/average cases.
- For file structures, be ready to compute blocking factor, number of blocks, I/O cost for search with/without index.

SAMPLE PROBLEMS (practice)

1. Given array of keys build BST and show inorder, then delete a node with two children (show step).
2. Construct Huffman codes for chars with frequencies: {A:45,B:13,C:12,D:16,E:9,E:5}.
3. Given adjacency list, run BFS from node s and produce distances and parents.
4. On an array, perform quicksort with median-of-3 pivot selection; show partition steps.
5. Given record size 300 bytes and block size 2048 bytes, compute blocking factor and number of blocks for 10,000 records.
6. Implement chaining hash table and demonstrate insert/search/delete for a set of keys.