



Gurukul Technicals Institute

Computer education · Robotics · Science projects · Delhi

FREE EBOOK

PROGRAMMING AND WEB

Data Structures Made Simple

Arrays, stacks, queues, linked lists, trees and graphs with everyday examples

INSIDE THIS BOOK

- Why data structures?
- Arrays
- Strings
- Stacks
- Queues and circular queues
- ...and 6 more chapters, practice questions and a glossary

Intermediate

College and diploma students

engineergurukul.com

+91 97116 99511

Data Structures Made Simple

Understand how programs organise data, from arrays and strings to stacks, queues, linked lists, trees, heaps, graphs and hash tables. Every idea starts with an everyday example, working C code shows how it is built, and a final chapter helps you choose the right structure for a job.

WHO IT IS FOR	College and diploma students
LEVEL	Intermediate
SUBJECT AREA	Programming and web
WHAT IS INSIDE	11 chapters, practice questions with answers, and a glossary

CONTENTS

1	Why data structures?	4
2	Arrays	5
3	Strings	7
4	Stacks	8
5	Queues and circular queues	11
6	Linked lists	13
7	Trees, binary search trees and heaps	17
8	Graphs: BFS and DFS	20
9	Hashing	21
10	Sorting and searching	22
11	Choosing the right structure	25
	Practice questions	27

Answers**27****Glossary****28**

Free ebook: you may read, print and share it for learning; it may not be sold or changed. © 2026 Gurukul Technicals Institute. First edition, October 2026. Contact details are on the back cover.

Why data structures?

Picture two shops with the same thousand books. In one, the books lie in a heap on the floor. In the other, they stand on shelves sorted by subject. Only the second can find a book in a minute. A program faces the same problem with its data. A **data structure** is a way of arranging data in memory so that the jobs you need to do with it are quick and simple. Choosing well often matters more than clever code.

You already use data structures in daily life:

Everyday thing	Data structure	Why it fits
Plates stacked in a canteen	Stack	You take the top plate, the last one added
Ticket queue at a railway counter	Queue	The first person in line is served first
A train of coaches joined by couplings	Linked list	Each coach is connected to the next one
A family tree	Tree	Each person branches into children
A metro map	Graph	Stations are joined by lines in many directions
A dictionary with thumb tabs	Hash table	The first letter takes you straight to the right part

Operations

Every structure is asked to do the same few jobs, called **operations**: insert, delete, search, traverse (visit every item once) and sort. A structure that makes one operation fast often makes another slow. Arrays are fast to read by position but slow to insert into, and linked lists are the opposite.

Measuring speed with Big-O

We compare structures by how the work grows as the data grows, written in **Big-O notation**. $O(1)$ means the work stays the same however much data there is. $O(n)$ means it grows in step with n , the number of items. $O(\log n)$ grows very slowly, and $O(n^2)$ very fast.

REMEMBER

There is no "best" data structure. There is only the best one for the operations your program does most often.

All code in this book is standard C (C99 or later). Compile it with `gcc file.c -o file`, then run it.

CHAPTER 2

Arrays

An **array** stores items of one type in neighbouring memory cells. Each item is reached by its **index**, its position number, which starts at 0 in C.

```
index:   0    1    2    3    4
         +---+---+---+---+---+
marks    | 45 | 78 | 62 | 90 | 55 |
         +---+---+---+---+---+
```

Because the cells are side by side, the computer finds any item with one sum: address of item i = starting address + $i \times$ size of one item. So reading `marks[3]` takes the same time in an array of 5 items or 5 lakh: $O(1)$.

Inserting and deleting means shifting

The price is paid when the array changes. To insert 70 at index 2, every item from index 2 onwards must move one place right to make room. To delete an item, everything after it moves one place left to close the gap.

C

```
#include <stdio.h>

void print(int a[], int n) {
    for (int i = 0; i < n; i++)
        printf("%d ", a[i]);
    printf("\n");
}

int main(void) {
    int marks[10] = {45, 78, 62, 90, 55};
    int n = 5;

    /* insert 70 at index 2: shift right first */
    for (int i = n; i > 2; i--)
        marks[i] = marks[i - 1];
    marks[2] = 70;
    n++;
    print(marks, n);    /* 45 78 70 62 90 55 */

    /* delete the item at index 4: shift left */
    for (int i = 4; i < n - 1; i++)
        marks[i] = marks[i + 1];
    n--;
    print(marks, n);    /* 45 78 70 62 55 */
    return 0;
}
```

Inserting or deleting at the front moves all n items, so these operations are $O(n)$. The array was declared with room for 10 though it starts with 5, because a C array cannot grow after it is created.

SAFETY

C does not check indexes. Writing to `marks[10]` in a 10-item array silently overwrites other memory and can crash the program or produce strange results. Always keep indexes from 0 to size minus 1.

Two-dimensional arrays

Marks for 30 students in 5 subjects fit a **two-dimensional array**: `int marks[30][5]`; . Here `marks[2][4]` is the third student's fifth subject. C stores the table row by row, called **row-major order**.

TIP

Use an array when you know roughly how many items there will be and mostly read them by position.

Strings

In C, a **string** is an array of characters that ends with a special character, `'\0'`, called the **null character**. The word "Delhi" needs six cells: five letters plus the `'\0'`, which tells every string function where the text stops.

```
+---+---+---+---+---+
| D | e | l | h | i | \0 |
+---+---+---+---+---+
```

Forget room for `'\0'` and string functions keep reading into memory that is not yours.

Useful functions from string.h

Function	What it does	Example result
<code>strlen(s)</code>	counts characters before <code>'\0'</code>	<code>strlen("Delhi")</code> is 5
<code>strcpy(d, s)</code>	copies s into d	d becomes a copy of s
<code>strcat(d, s)</code>	adds s to the end of d	"Pune" + "city" gives "Punecity"
<code>strcmp(a, b)</code>	compares two strings	0 if they are equal

Comparing strings with `==` does not compare the letters. It only checks whether two addresses are the same. Use `strcmp` and test for 0.

Example: a palindrome check

A **palindrome** reads the same forwards and backwards, like "malayalam" or "level". The idea is to keep two indexes, one at each end, and walk them towards each other.

C

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

int is_palindrome(const char s[]) {
    int i = 0;
    int j = (int)strlen(s) - 1;
    while (i < j) {
        if (s[i] != s[j])
            return 0;
        i++;
        j--;
    }
    return 1;
}

int main(void) {
    char word[] = "malayalam";
    printf("%s has %zu letters\n", word, strlen(word));
    printf("Palindrome? %s\n",
        is_palindrome(word) ? "yes" : "no");
    return 0;
}
```

This loop does at most $n/2$ comparisons, so it is $O(n)$.

SAFETY

Never use the old `gets` function to read a string. It cannot know the size of your array and will overflow it on long input, which is why it was removed from the C language in the 2011 standard. Use `fgets(buffer, sizeof buffer, stdin)` instead.

TRY THIS

Change the program so that it ignores capital letters, so that "Malayalam" also counts. The function `tolower` from `ctype.h` will help.

CHAPTER 4

Stacks

A **stack** is a pile where you can only add or remove at the top. The last item put on is the first one taken off. This rule is called **LIFO**, short for Last In, First Out. Think of plates in a wedding buffet or bangles slid onto a stand.

Operation	Meaning	Speed
push	put an item on top	$O(1)$

Operation	Meaning	Speed
pop	remove and return the top item	O(1)
peek	look at the top item without removing it	O(1)

Pushing onto a full stack is **overflow**, and popping from an empty one is **underflow**. Good code checks for both.

Where stacks are used

- **Undo:** an editor pushes each change. Ctrl+Z pops the latest one and reverses it.
- **Back button:** a browser pushes each page you visit, and Back pops the previous one.
- **Function calls:** the computer keeps unfinished function calls on a "call stack" and returns to them in reverse order.
- **Checking brackets:** compilers check that every `(`, `[` and `{` is closed in the right order.

Example: a bracket checker

The rule is simple. Push every opening bracket. When a closing bracket arrives, pop the top item, which must be its matching opening bracket. At the end, the stack must be empty.

```

#include <stdio.h>
#define MAX 100

char stack[MAX];
int top = -1;          /* -1 means empty */

void push(char c) {
    if (top == MAX - 1) {
        printf("Stack overflow\n");
        return;
    }
    stack[++top] = c;
}

char pop(void) {
    if (top == -1)
        return '\0';    /* underflow: nothing to pop */
    return stack[top--];
}

int matches(char open, char close) {
    return (open == '(' && close == ')') ||
           (open == '[' && close == ']') ||
           (open == '{' && close == '}');
}

int balanced(const char *s) {
    top = -1;
    for (int i = 0; s[i] != '\0'; i++) {
        char c = s[i];
        if (c == '(' || c == '[' || c == '{')
            push(c);
        else if (c == ')' || c == ']' || c == '}') {
            if (!matches(pop(), c))
                return 0;
        }
    }
    return top == -1;    /* nothing left open */
}

int main(void) {
    printf("%d\n", balanced("{a[b(c)d]e}")); /* 1 */
    printf("%d\n", balanced("(a[b]c)"));    /* 0 */
    printf("%d\n", balanced("((a)"));       /* 0 */
    return 0;
}

```

Trace "(a[b]c)": push (, push [, then) arrives. The top is [, which does not match, so the answer is 0.

DID YOU KNOW

`top` starts at -1 so that the first push goes into index 0. Starting it at 0 leaves index 0 unused and breaks the empty test. A dry run on paper catches this.

CHAPTER 5

Queues and circular queues

A **queue** is a line where items join at the back, called the **rear**, and leave from the front. The first item in is the first item out. This rule is called **FIFO**, short for First In, First Out. A ticket counter, a doctor's waiting room with token numbers and a printer handling documents one by one all work this way.

Adding at the rear is called **enqueue**. Removing from the front is called **dequeue**. Both should be $O(1)$.

The problem with a simple array queue

Store a queue in an array of 5 slots. After five people join and two leave, slots 0 and 1 are empty, but the rear has reached the end of the array, so the queue says "full". Shifting everyone forward after each dequeue fixes this, but makes dequeue $O(n)$.

The circular queue

A **circular queue** treats the array as if its end were joined to its start, like seats around a round table. When the rear reaches the last slot, the next item goes into slot 0 if it is free. The remainder operator `%` makes the wrap-around easy: the slot after `i` is `(i + 1) % SIZE`.

C

```

#include <stdio.h>
#define SIZE 5

int q[SIZE];
int front = 0, count = 0;

int enqueue(int x) {
    if (count == SIZE)
        return 0;
    q[(front + count) % SIZE] = x;
    count++;
    return 1;
}

int dequeue(int *x) {
    if (count == 0)
        return 0;
    *x = q[front];
    front = (front + 1) % SIZE;
    count--;
    return 1;
}

int main(void) {
    int x;
    for (int t = 101; t <= 105; t++)
        enqueue(t);
    dequeue(&x);
    printf("Served %d\n", x);
    dequeue(&x);
    printf("Served %d\n", x);
    enqueue(106);
    enqueue(107);
    printf("Room for 108? %s\n", enqueue(108) ? "yes" : "no");
    while (dequeue(&x))
        printf("%d ", x);
    printf("\n");
    return 0;
}

```

This version keeps `front` and a `count` of items instead of a rear index. The rear slot is always `(front + count) % SIZE`.

EXAMPLE

A clinic in Chennai shows five waiting tokens on a screen. After tokens 101 and 102 go in, tokens 106 and 107 take the freed places, slots 0 and 1, just as in the program above.

A **deque** (double-ended queue, said "deck") allows adding and removing at both ends. A **priority queue** serves the most important item first, like a hospital emergency ward, and is usually built with a heap.

REMEMBER

Stack = LIFO, add and remove at the same end. Queue = FIFO, add at one end and remove at the other.

CHAPTER 6

Linked lists

An array keeps items side by side, which makes insertion slow. A **linked list** keeps each item in its own small block of memory, called a **node**, anywhere in memory. Each node holds its data and a **pointer** (the memory address) to the next node. The first node is called the **head**. The last node's pointer is `NULL`, meaning "nothing comes next".

```
head
|
v
[10 | *]-->[20 | *]-->[30 | NULL]
```

Like coaches of a train, nodes can be added or removed by changing the links, without moving any other node.

A singly linked list in C

Nodes are made while the program runs with `malloc`, which asks the system for memory, and returned with `free`.

```

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

struct Node {
    int data;
    struct Node *next;
};

struct Node *new_node(int value) {
    struct Node *n = malloc(sizeof *n);
    if (n == NULL) {
        printf("Out of memory\n");
        exit(1);
    }
    n->data = value;
    n->next = NULL;
    return n;
}

void push_front(struct Node **head, int value) {
    struct Node *n = new_node(value);
    n->next = *head;
    *head = n;
}

int delete_value(struct Node **head, int value) {
    struct Node *p = *head, *prev = NULL;
    while (p != NULL && p->data != value) {
        prev = p;
        p = p->next;
    }
    if (p == NULL)
        return 0; /* not found */
    if (prev == NULL)
        *head = p->next; /* it was the first node */
    else
        prev->next = p->next; /* step over it */
    free(p);
    return 1;
}

void print_list(struct Node *p) {
    while (p != NULL) {
        printf("%d -> ", p->data);
        p = p->next;
    }
    printf("NULL\n");
}

void free_list(struct Node *p) {
    while (p != NULL) {
        struct Node *next = p->next;
        free(p);
    }
}

```

```

        p = next;
    }
}

int main(void) {
    struct Node *head = NULL;
    push_front(&head, 30);
    push_front(&head, 20);
    push_front(&head, 10);
    print_list(head);      /* 10 -> 20 -> 30 -> NULL */
    delete_value(&head, 20);
    print_list(head);      /* 10 -> 30 -> NULL */
    free_list(head);
    return 0;
}

```

The functions take `struct Node **head`, a pointer to the head pointer, because adding at the front or deleting the first node changes `head` itself. To delete 20, the code walks along with `prev` one step behind, points the previous node's `next` past 20, then frees it.

Arrays and linked lists compared

Job	Array	Linked list
Read item number i	$O(1)$	$O(n)$, you must walk from the head
Insert or delete at the front	$O(n)$, shifting	$O(1)$
Insert after a node you already hold	$O(n)$	$O(1)$
Grow while the program runs	No, size is fixed	Yes
Extra memory	None	One pointer per node

Doubly linked lists

In a **doubly linked list**, each node also points back to the previous node, so you can walk both ways and delete a node without searching for the one before it. The cost is an extra pointer per node.

```

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

struct DNode {
    int data;
    struct DNode *prev, *next;
};

void push_front(struct DNode **head, int value) {
    struct DNode *n = malloc(sizeof *n);
    if (n == NULL)
        exit(1);
    n->data = value;
    n->prev = NULL;
    n->next = *head;
    if (*head != NULL)
        (*head)->prev = n;
    *head = n;
}

int main(void) {
    struct DNode *head = NULL, *p, *last = NULL;
    for (int v = 3; v >= 1; v--)
        push_front(&head, v);
    for (p = head; p != NULL; p = p->next) {
        printf("%d ", p->data);          /* 1 2 3 */
        last = p;
    }
    printf("\n");
    for (p = last; p != NULL; p = p->prev)
        printf("%d ", p->data);          /* 3 2 1 */
    printf("\n");
    while (head != NULL) {
        p = head->next;
        free(head);
        head = p;
    }
    return 0;
}

```

A **circular linked list** joins the last node back to the first. It suits jobs that go round and round, such as giving players turns in a game.

SAFETY

Every `malloc` needs a matching `free`. Memory you forget to free is a **memory leak**, and in a program that runs for days, leaks slowly eat up all the memory.

Trees, binary search trees and heaps

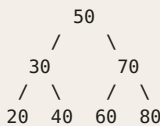
A **tree** stores items in levels, like a family tree or the folders on a computer. Some words you need:

- **Root:** the top node, which has no parent.
- **Parent and child:** a node directly above another is its parent.
- **Leaf:** a node with no children.
- **Subtree:** a node with everything below it.

A **binary tree** is a tree where each node has at most two children, called left and right.

Binary search trees

A **binary search tree** (BST) is a binary tree with one rule: for every node, all keys in its left subtree are smaller and all keys in its right subtree are larger. Inserting 50, 30, 70, 20, 40, 60, 80 gives this tree:



To find 60, start at 50 and go right (60 is larger), then left from 70: found. Each step throws away about half the tree, so a search in a balanced tree is $O(\log n)$.

Traversals

A **traversal** visits every node once. There are three common orders, named after when the root is visited:

Traversal	Order	Result for the tree above
Inorder	left, root, right	20 30 40 50 60 70 80
Preorder	root, left, right	50 30 20 40 70 60 80
Postorder	left, right, root	20 40 30 60 80 70 50

Inorder traversal of a BST always gives the keys in sorted order.

```

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

struct TNode {
    int key;
    struct TNode *left, *right;
};

struct TNode *insert(struct TNode *root, int key) {
    if (root == NULL) {
        root = malloc(sizeof *root);
        if (root == NULL)
            exit(1);
        root->key = key;
        root->left = root->right = NULL;
    } else if (key < root->key) {
        root->left = insert(root->left, key);
    } else if (key > root->key) {
        root->right = insert(root->right, key);
    }
    return root;          /* duplicates are ignored */
}

int search(struct TNode *root, int key) {
    while (root != NULL) {
        if (key == root->key)
            return 1;
        root = (key < root->key) ? root->left : root->right;
    }
    return 0;
}

void inorder(struct TNode *r) {
    if (r == NULL)
        return;
    inorder(r->left);
    printf("%d ", r->key);
    inorder(r->right);
}

void free_tree(struct TNode *r) {
    if (r == NULL)
        return;
    free_tree(r->left);
    free_tree(r->right);
    free(r);
}

int main(void) {
    int keys[] = {50, 30, 70, 20, 40, 60, 80};
    struct TNode *root = NULL;
    for (int i = 0; i < 7; i++)
        root = insert(root, keys[i]);
}

```

```

    inorder(root);                      /* 20 30 40 50 60 70 80 */
    printf("\n");
    printf("40: %d, 65: %d\n",
           search(root, 40), search(root, 65));
    free_tree(root);
    return 0;
}

```

Preorder and postorder are the same function with the `printf` moved before or after the two calls. In fact `free_tree` is a postorder traversal: it frees both children before their parent.

REMEMBER

If you insert keys that are already sorted, such as 10, 20, 30, 40, every node goes to the right. The "tree" becomes a long chain and search falls to $O(n)$. Self-balancing trees, such as AVL trees and red-black trees, fix this by rearranging nodes as they go.

Heaps: the idea

A **max-heap** is a binary tree where every parent is greater than or equal to its children, so the largest item is always at the root. (A **min-heap** keeps the smallest at the root.) A heap is always filled level by level, left to right, so it fits neatly in an array with no gaps. For the item at index i , counting from 0:

Relative	Index
Left child	$2i + 1$
Right child	$2i + 2$
Parent	$(i - 1) / 2$, rounded down

The array 90, 70, 80, 30, 50, 60 is a max-heap. To **insert** 85, put it at the end and swap it upward while it beats its parent: it passes 80 and stops under 90, giving 90, 70, 85, 30, 50, 60, 80. To **remove the largest**, take the root, move the last item to the top and swap it down with its larger child until it fits. Both are $O(\log n)$.

DID YOU KNOW

Heaps also give **heap sort**: build a heap, then remove the largest item again and again. It is $O(n \log n)$ and needs no extra array.

Graphs: BFS and DFS

A **graph** is a set of points, called **vertices**, joined by lines, called **edges**. Unlike a tree, a graph can have many routes between two points, and routes that lead back to where they started. Road maps, metro networks and friend lists are all graphs.

- In an **undirected** graph, an edge works both ways, like a two-way road.
- In a **directed** graph, each edge has a direction, like a one-way street or "A follows B".
- In a **weighted** graph, each edge carries a number, such as the distance in kilometres.

Storing a graph

An **adjacency matrix** is a table with a row and a column per vertex, and 1 where an edge exists. It answers "are A and B joined?" in O(1) but needs $n \times n$ cells. An **adjacency list** keeps a list of neighbours for each vertex, which saves memory when each point has only a few links.

Here is a small town as an adjacency list. H is home, S school, M market, L library, P park and T temple.

Vertex	Neighbours
H	S, M
S	H, L
M	H, L, P
L	S, M, T
P	M
T	L

Breadth-first search (BFS)

BFS explores the graph in rings: first every place one step from the start, then every place two steps away, and so on. It uses a queue and marks places as "seen" so that none is visited twice.

1. Mark the start as seen and put it in the queue.
2. Take the place at the front of the queue and visit it.
3. Put each of its unseen neighbours at the back of the queue, and mark them seen.
4. Repeat steps 2 and 3 until the queue is empty.

Visit	Queue afterwards
H	S, M

Visit	Queue afterwards
S	M, L
M	L, P
L	P, T
P	T
T	empty

The visiting order is H, S, M, L, P, T. Because BFS goes ring by ring, it finds the route with the fewest edges: the temple is 3 steps from home (H, S, L, T).

Depth-first search (DFS)

DFS goes as deep as it can along one path and backs up only at a dead end, like exploring a maze. It uses a stack, or a function that calls itself (**recursion**). From H, taking neighbours in the listed order, DFS visits H, S, L, M, P, then backs up to L and visits T.

TIP

Use BFS for "fewest steps" questions and DFS to explore every path, as in a maze puzzle. Both visit each vertex and edge once: $O(V + E)$, for V vertices and E edges.

CHAPTER 9

Hashing

Searching a sorted array takes $O(\log n)$. What if we could jump straight to an item, in $O(1)$, most of the time? That is the promise of **hashing**.

A **hash table** is an array of slots. A **hash function** turns each key, such as a roll number, into a slot number. A simple hash function for whole numbers is the remainder after dividing by the table size: `slot = key % 10` for a table of 10 slots.

Key	key % 10	Slot
23	3	3
45	5	5
12	2	2
50	0	0

To find roll number 45 later, work out $45 \% 10 = 5$ and look in slot 5. There is no searching at all.

Collisions

Now insert 33. Its slot is also 3, which 23 already holds. Two keys landing in the same slot is called a **collision**. Collisions cannot be avoided completely, so every hash table needs a plan:

- **Chaining:** each slot holds a small linked list. Slot 3 becomes 23 then 33. To search, go to the slot and walk its short list.
- **Linear probing:** if the slot is taken, try the next one, then the next, wrapping round at the end. 33 goes into slot 4.

Keeping it fast

The **load factor** is the number of stored items divided by the number of slots. As it rises, collisions multiply, so real hash tables grow, often doubling their slots, when it passes a set limit. A good hash function also spreads keys evenly: % 10 on keys that all end in 0 would put every key in slot 0.

EXAMPLE

A library in Bhopal stores 2,000 members by membership number in a hash table of 4,000 slots. The load factor is 0.5, so most lookups go straight to the right slot.

Search, insert and delete are $O(1)$ on average. The worst case, $O(n)$, needs many keys to collide, which a good hash function makes very unlikely.

REMEMBER

A hash table keeps no order. If you need items in sorted order, or "all roll numbers between 20 and 40", use a sorted array or a balanced search tree instead.

CHAPTER 10

Sorting and searching

Sorted data is easier to search and compare. Here are five classic methods, sorting 29, 10, 14, 37, 13 into increasing order.

- **Bubble sort:** compare neighbours and swap them if they are in the wrong order. After each pass the largest remaining item has moved to the end. Pass 1 gives 10, 14, 29, 13, 37.
- **Selection sort:** swap the smallest item into first place, then the smallest of the rest into second place, and so on. Step 1 gives 10, 29, 14, 37, 13.
- **Insertion sort:** slide each item left into its place among the items already sorted, like arranging playing cards in your hand.
- **Merge sort:** split the list in half, sort each half, then merge the halves by repeatedly taking the smaller front item.

- **Quick sort:** pick a **pivot**, move smaller items to its left and larger ones to its right, then sort each side the same way.

Here is quick sort in C, with the last item as the pivot, followed by binary search.

```

#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];          /* the last item */
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (a[j] < pivot) {
            i++;
            swap(&a[i], &a[j]);
        }
    }
    swap(&a[i + 1], &a[high]);
    return i + 1;                /* pivot's final place */
}

void quick_sort(int a[], int low, int high) {
    if (low < high) {
        int p = partition(a, low, high);
        quick_sort(a, low, p - 1);
        quick_sort(a, p + 1, high);
    }
}

int binary_search(const int a[], int n, int key) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (a[mid] == key)
            return mid;
        if (a[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;                    /* not found */
}

int main(void) {
    int a[] = {29, 10, 14, 37, 13};
    quick_sort(a, 0, 4);
    for (int i = 0; i < 5; i++)
        printf("%d ", a[i]);      /* 10 13 14 29 37 */
    printf("\n");
    printf("37 is at index %d\n", binary_search(a, 5, 37));
    printf("15 is at index %d\n", binary_search(a, 5, 15));
}

```

```
    return 0;
}
```

Comparing the five

Method	Best	Average	Worst	Extra memory	Stable
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes
Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	No

Bubble sort's $O(n)$ best case assumes it stops after a pass with no swaps. A **stable** sort keeps items with equal keys in their original order, so students with equal marks stay in alphabetical order.

TIP

Insertion sort is very fast on small or nearly sorted lists. Quick sort is usually fastest in practice, but hits its worst case when a poor pivot is picked every time. With the last item as the pivot, that happens on an already sorted list.

Searching

Linear search checks items one by one and works on any list: $O(n)$. **Binary search** needs a sorted array and halves the part left to search at each step: $O(\log n)$, at most 20 comparisons for 10 lakh items. In the code, $\text{low} + (\text{high} - \text{low}) / 2$ equals $(\text{low} + \text{high}) / 2$ but cannot overflow on huge indexes.

CHAPTER 11

Choosing the right structure

In real work the question is always: which structure fits this job? Start with three questions.

1. Which operations happen most: reading by position, searching, inserting, deleting, or taking the newest, oldest or most important item?
2. Must the data stay in order?
3. Is the amount of data known in advance?

A quick-reference table

Structure	Strong at	Weak at
Array	reading by index, $O(1)$	inserting or deleting in the middle, $O(n)$
Linked list	inserting or deleting at a known node, $O(1)$	reading by position, $O(n)$
Stack	newest item first, $O(1)$	reaching anything below the top
Queue	oldest item first, $O(1)$	reaching the middle
Binary search tree	sorted order and search, $O(\log n)$ if balanced	becomes $O(n)$ when unbalanced
Heap	largest or smallest item, $O(1)$ to see, $O(\log n)$ to remove	searching for any other item
Hash table	search by key, $O(1)$ on average	anything needing order
Graph	connections and routes	needs more memory and code than simpler structures

Worked choices

EXAMPLE

A railway reservation system keeps a waiting list. Whoever booked first is confirmed first when a seat frees up. That is first in, first out, so a **queue**.

- A drawing app's undo button needs the most recent change first: a **stack**.
- A phone's contact list must stay in alphabetical order while names are added and removed: a **balanced search tree**, or a sorted array if names rarely change.
- A booking site looks up a ticket by its ten-digit number among crores of tickets: a **hash table**.
- A hospital emergency desk always treats the most serious patient next: a **priority queue** built on a heap.
- A delivery app plans routes between shops and homes across Bengaluru: a **graph**, searched with BFS or a related method.

TRY THIS

For each of these, name a structure and give your reason: the "recently closed tabs" list in a browser, a printer shared by a whole office, and finding whether two people on a social network are connected through friends.

Often the answer is two structures working together, such as a graph search with a queue inside it.

Practice questions

1. What is a data structure? Give two everyday examples.
2. An `int` takes 4 bytes and an array starts at address 1000. What is the address of `a[5]`?
3. Why is inserting at the front of an array $O(n)$?
4. How many characters of memory does the C string "Kolkata" need, and why?
5. Explain LIFO and FIFO with one example each.
6. Show the stack after each step: push 5, push 8, pop, push 3, push 9, pop.
7. Using the bracket rule, is `"[(a+b)*{c}]"` balanced? Is `"(a+b]"` balanced?
8. In a circular queue of size 5, front is 3 and count is 4. In which slot will the next item go?
9. Write the steps to insert a new node at the front of a singly linked list.
10. Insert 40, 20, 60, 10, 30, 50 into an empty BST. Write the inorder, preorder and postorder traversals.
11. Why can a BST become slow, and what is one fix?
12. In a heap stored in an array from index 0, what are the indexes of the children and the parent of index 4?
13. Insert 95 into the max-heap 90, 70, 80, 30, 50, 60. Show the array after the insert.
14. Do BFS from L on the town graph in the Graphs chapter, taking neighbours in the listed order. Write the visiting order.
15. Using `slot = key % 7`, place 15, 22, 8 and 29 into a table of 7 slots with linear probing.
16. What is a stable sort? Name two stable sorts and one that is not stable.
17. Give the worst-case Big-O of quick sort and of merge sort. When does quick sort hit its worst case with the last item as the pivot?
18. Choose a data structure for each, with a reason: (a) the "back" button of a browser, (b) storing a fixed list of 12 monthly sales figures, (c) finding a student by Aadhaar-style unique ID among lakhs of records.

ANSWER KEY

Answers

1. A way of organising data in memory so that the needed operations are quick and simple.
Examples: a canteen stack of plates (stack) and a ticket line (queue).
2. $1000 + 5 \times 4 = 1020$.
3. Every existing item must shift one place right to make room, so all n items move.

4. 8 characters: 7 letters plus the null character `'\0'` that marks the end.
5. LIFO, last in first out: a stack of plates. FIFO, first in first out: a queue at a ticket counter.
6. [5], then [5, 8], then pop 8 to get [5], then [5, 3], then [5, 3, 9], then pop 9 to get [5, 3].
7. The first is balanced: every bracket closes in the right order and the stack ends empty. The second is not: `]` arrives while the top of the stack is `(`.
8. Slot $(3 + 4) \% 5 = 2$.
9. Create a new node and store the data in it. Set its `next` to the current head. Make head point to the new node.
10. The tree has 40 at the root, 20 and 60 as its children, 10 and 30 under 20, and 50 under 60. Inorder: 10 20 30 40 50 60. Preorder: 40 20 10 30 60 50. Postorder: 10 30 20 50 60 40.
11. Inserting keys in sorted order makes every node go to one side, so the tree becomes a chain and search becomes $O(n)$. Self-balancing trees such as AVL or red-black trees fix this.
12. Children at $2 \times 4 + 1 = 9$ and $2 \times 4 + 2 = 10$. Parent at $(4 - 1) / 2 = 1$, rounded down.
13. Put 95 at index 6. Its parent is 80 at index 2, so swap. Its new parent is 90 at index 0, so swap again. The result is 95, 70, 90, 30, 50, 60, 80.
14. L, S, M, T, H, P. From L the queue gets S, M and T. S adds H. M adds P.
15. 15 goes to slot 1. $22 \% 7 = 1$ is taken, so it goes to slot 2. $8 \% 7 = 1$ is taken and so is 2, so it goes to slot 3. $29 \% 7 = 1$, and slots 1, 2 and 3 are taken, so it goes to slot 4.
16. A sort that keeps equal keys in their original order. Insertion sort and merge sort are stable. Selection sort and quick sort usually are not.
17. Quick sort $O(n^2)$, merge sort $O(n \log n)$. Quick sort hits its worst case when the list is already sorted, because each pivot is then the largest item left and every other item lands on one side.
18. (a) A stack, because the most recent page is needed first. (b) An array, because the size is fixed and items are read by month number. (c) A hash table keyed by the ID, for $O(1)$ average lookups.

WORD LIST

Glossary

Array

items of one type stored in neighbouring memory cells and reached by index.

Big-O notation

a way of describing how the work of an operation grows as the data grows.

Binary search tree

a binary tree where every left subtree holds smaller keys and every right subtree larger ones.

Breadth-first search (BFS)

a graph search that visits places ring by ring, using a queue.

Circular queue

a queue in an array whose end wraps round to its start.

Collision

two keys being sent to the same slot by a hash function.

Depth-first search (DFS)

a graph search that follows one path as deep as possible before backing up.

Graph

a set of vertices joined by edges.

Hash function

a rule that turns a key into a slot number in a hash table.

Hash table

an array of slots where items are placed by a hash function for fast lookup.

Heap

a tree kept in an array where each parent is at least as large (max-heap) or as small (min-heap) as its children.

Index

the position number of an item in an array, starting from 0 in C.

Linked list

a chain of nodes where each node points to the next.

Memory leak

memory taken with malloc and never returned with free.

Node

one block in a linked list, tree or graph, holding data and links.

Pointer

a variable that holds a memory address.

Queue

a first-in, first-out structure: add at the rear, remove from the front.

Recursion

a function solving a problem by calling itself on a smaller part of it.

Stable sort

a sort that keeps items with equal keys in their original order.

Stack

a last-in, first-out structure where items are added and removed at the top.



Gurukul Technicals Institute

Computer education institute in Delhi, with authorised member centres, robotics classes and science projects, working models and DIY kits.

COURSES

Computer & Office Skills · Accounting & Business Software · Programming & Web Development · Design & Digital Skills · Hardware & Networking · Robotics & Coding · Other Programmes

PROJECTS AND KITS

Ready-made school projects · DIY and STEM kits · Robotics and coding · Components and spares · Engineering projects

CONTACT

Head office: A-10, Chief Commissioner Colony, opposite Rana Pratap Bagh, Delhi 110007

Phone: 011 2721 1512 · **Mobile and WhatsApp:** +91 97116 99511

Email: engineergurukul@yahoo.co.in

Website: engineergurukul.com · **Instagram:** [@science_projects_club](https://www.instagram.com/science_projects_club)

More free ebooks for computer courses, robotics and science projects: engineergurukul.com/resources/ebooks/

Free to read, print and share. Not for sale.