**Recursion fundamentals**

**Concept — Base case and recursive call**

* **Recursion**: a function that calls itself to solve a smaller instance of the same problem.
* **Base case**: the condition(s) where the function returns directly without calling itself. Prevents infinite recursion.
* **Recursive call**: when the function calls itself with a smaller/closer-to-base argument.
* Each recursive call creates a new *stack frame* containing parameters and local variables. Frames are pushed on the call stack; they pop when the call returns.

**Why both are necessary:** Without a base case recursion never ends. Without recursive calls you have no recursion.

**Program: Factorial — recursive vs iterative**

**File:** FactorialRecursive.java

// FactorialRecursive.java

public class FactorialRecursive {

// Recursive factorial

public static long factorialRecursive(int n) {

if (n < 0) throw new IllegalArgumentException("n must be >= 0");

if (n == 0 || n == 1) return 1; // <-- BASE CASE

return n \* factorialRecursive(n - 1); // <-- RECURSIVE CALL

}

// Iterative factorial for comparison

public static long factorialIterative(int n) {

if (n < 0) throw new IllegalArgumentException("n must be >= 0");

long res = 1;

for (int i = 2; i <= n; i++) res \*= i;

return res;

}

public static void main(String[] args) {

System.out.println("Recursive 5! = " + factorialRecursive(5));

System.out.println("Iterative 5! = " + factorialIterative(5));

}

}

**Program explanation (line-by-line / block):**

* factorialRecursive:
  + Check invalid input.
  + if (n == 0 || n == 1) return 1; — *base case*; stops recursion.
  + return n \* factorialRecursive(n - 1); — multiply n by factorial of smaller n.
* factorialIterative shows an iterative loop doing the same work.
* Complexity: both produce O(n) time; recursive uses O(n) stack frames (space), iterative uses O(1) extra space.

**2. Recursive vs Iterative approaches (and when to use which)**

**Concept**

* **Recursive** code often maps directly to the problem (trees, divide & conquer) and can be simpler/cleaner.
* **Iterative** approaches usually use explicit loops and often use less memory (no call-stack overhead).
* **Tail recursion** can be optimized by some languages into iterative loops — Java does **not** reliably do tail-call optimization.
* Choose recursion for clarity (trees, recursion-friendly problems) and iteration when memory/deep recursion is a concern or performance matters.

**Program: Fibonacci — naive recursive, memoized, iterative**

**File:** FibonacciVariants.java

// FibonacciVariants.java

import java.util.Arrays;

public class FibonacciVariants {

// Naive recursion (exponential time)

public static long fibRecursive(int n) {

if (n <= 1) return n;

return fibRecursive(n - 1) + fibRecursive(n - 2);

}

// Memoized recursion (top-down DP) — O(n)

public static long fibMemo(int n) {

if (n < 0) throw new IllegalArgumentException("n >= 0 required");

long[] memo = new long[n + 1];

Arrays.fill(memo, -1);

return fibMemoHelper(n, memo);

}

private static long fibMemoHelper(int n, long[] memo) {

if (n <= 1) return n;

if (memo[n] != -1) return memo[n];

memo[n] = fibMemoHelper(n - 1, memo) + fibMemoHelper(n - 2, memo);

return memo[n];

}

// Iterative (bottom-up) — O(n) time, O(1) space

public static long fibIterative(int n) {

if (n <= 1) return n;

long a = 0, b = 1;

for (int i = 2; i <= n; i++) {

long c = a + b;

a = b; b = c;

}

return b;

}

public static void main(String[] args) {

int n = 20;

System.out.println("fibRecursive(" + n + ") = " + fibRecursive(10) + " (use small n for naive recursion)");

System.out.println("fibMemo(" + n + ") = " + fibMemo(n));

System.out.println("fibIterative(" + n + ") = " + fibIterative(n));

}

}

**Program explanation:**

* fibRecursive: simple but exponential time O(2^n) because it recomputes subproblems.
* fibMemo: saves results in memo[] to avoid recomputation — O(n) time and O(n) space.
* fibIterative: uses two variables and runs in linear time O(n) with constant extra space.
* Use memoization or iterative DP if n grows; recursive naive only for tiny n or teaching.

**3. Stack ADT — LIFO, stack operations**

**Concept**

* **Stack ADT**: Last-In-First-Out (LIFO). The last pushed item is the first popped.
* Standard operations:
  + push(x) — put x on top.
  + pop() — remove and return top element.
  + peek() / top() — view the top without removing.
  + isEmpty(), size().
* Typical time complexity: O(1) for push/pop/peek.

**Why stacks?** They model nested, last-used resources: function call stack, undo history, bracket matching, DFS, expression evaluation.

**Program A: Array-based Stack (resizing)**

**File:** ArrayStack.java

// ArrayStack.java

import java.util.Arrays;

import java.util.EmptyStackException;

public class ArrayStack<T> {

private Object[] data;

private int size;

private static final int DEFAULT\_CAP = 8;

public ArrayStack() { this(DEFAULT\_CAP); }

public ArrayStack(int cap) {

data = new Object[Math.max(cap, DEFAULT\_CAP)];

size = 0;

}

public void push(T value) {

ensureCapacity(size + 1);

data[size++] = value;

}

@SuppressWarnings("unchecked")

public T pop() {

if (isEmpty()) throw new EmptyStackException();

T val = (T) data[--size];

data[size] = null; // help GC

return val;

}

@SuppressWarnings("unchecked")

public T peek() {

if (isEmpty()) throw new EmptyStackException();

return (T) data[size - 1];

}

public boolean isEmpty() { return size == 0; }

public int size() { return size; }

private void ensureCapacity(int minCap) {

if (minCap > data.length) {

int newCap = data.length \* 2;

if (newCap < minCap) newCap = minCap;

data = Arrays.copyOf(data, newCap);

}

}

// Demo

public static void main(String[] args) {

ArrayStack<Integer> st = new ArrayStack<>();

st.push(10); st.push(20); st.push(30);

System.out.println("Top: " + st.peek()); // 30

System.out.println("Pop: " + st.pop()); // 30

System.out.println("Pop: " + st.pop()); // 20

System.out.println("Size: " + st.size()); // 1

}

}

**Program explanation:**

* data is Object[] backing storage (generic array emulation).
* push calls ensureCapacity (doubles capacity when needed), stores element, increments size.
* pop decreases size, returns the element; sets array slot to null for GC.
* peek returns data[size-1].
* ensureCapacity uses Arrays.copyOf to resize — amortized O(1) push.
* Usage: typical stack behavior with constant-time operations.

**Program B: Linked-list-based Stack**

**File:** LinkedStack.java

// LinkedStack.java

import java.util.EmptyStackException;

public class LinkedStack<T> {

private static class Node<T> {

T data; Node<T> next;

Node(T d, Node<T> n) { data = d; next = n; }

}

private Node<T> top;

private int size;

public LinkedStack() { top = null; size = 0; }

public void push(T value) {

top = new Node<>(value, top);

size++;

}

public T pop() {

if (isEmpty()) throw new EmptyStackException();

T val = top.data;

top = top.next;

size--;

return val;

}

public T peek() {

if (isEmpty()) throw new EmptyStackException();

return top.data;

}

public boolean isEmpty() { return top == null; }

public int size() { return size; }

// Demo

public static void main(String[] args) {

LinkedStack<String> st = new LinkedStack<>();

st.push("A"); st.push("B"); st.push("C");

System.out.println("Peek: " + st.peek()); // C

while (!st.isEmpty()) System.out.println("Pop: " + st.pop());

}

}

**Program explanation:**

* Stack is a singly linked list with top pointing to head.
* push creates a new node and sets it as new top (O(1)).
* pop removes top node, returns its value (O(1)).
* This implementation never needs resizing, uses O(n) space proportional to elements.

**4. Applications of stack**

We’ll cover three classic stack applications: parentheses matching, infix → postfix (and evaluation), and visualization of the function call stack.

**4A Parentheses matching (balanced brackets)**

**Concept**

* Push opening brackets ((, [, {) onto a stack.
* When encountering a closing bracket, pop top and ensure it matches (e.g., popped ( matches )).
* If stack is empty when a closing bracket is found or stack is non-empty after processing, expression is unbalanced.

**Time:** O(n). **Space:** O(n) in worst case.

**Program: Parentheses matching (simple)**

**File:** ParenthesesMatching.java

// ParenthesesMatching.java

import java.util.ArrayDeque;

import java.util.Deque;

public class ParenthesesMatching {

public static boolean isBalanced(String s) {

Deque<Character> st = new ArrayDeque<>();

for (int i = 0; i < s.length(); i++) {

char c = s.charAt(i);

if (c == '(' || c == '[' || c == '{') st.push(c);

else if (c == ')' || c == ']' || c == '}') {

if (st.isEmpty()) return false;

char top = st.pop();

if (!matches(top, c)) return false;

}

}

return st.isEmpty();

}

private static boolean matches(char open, char close) {

return (open == '(' && close == ')') ||

(open == '[' && close == ']') ||

(open == '{' && close == '}');

}

public static void main(String[] args) {

String[] tests = {"([]){}", "([)]", "((1+2)\*3)", "([]", "([{}])"};

for (String t : tests) System.out.printf("%-10s : %s%n", t, isBalanced(t) ? "balanced" : "NOT balanced");

}

}

**Program explanation:**

* Uses ArrayDeque<Character> as a stack (push/pop).
* Iterate characters: push opens; on close check matches(pop(), close).
* At end, stack must be empty for balanced expression.
* Example outputs show which inputs are balanced.

**4B Infix → Postfix conversion (Shunting-yard) and Postfix evaluation**

**Concept (Shunting-yard)**

* Convert infix expression (normal notation, e.g. 3 + 4 \* 2) to postfix (Reverse Polish Notation) where operators follow operands (e.g. 3 4 2 \* +).
* Use an operator stack to manage precedence and associativity:
  + When token is number → output.
  + When token is operator → pop higher-or-equal precedence operators from stack to output (depending on associativity) then push current operator.
  + When ( push it; when ) pop until (.
* After processing all tokens, pop remaining operators to output.

**Evaluation of postfix**:

* Use a stack of numbers: push numbers; when operator appears pop two operands (b then a), compute a op b, push result; final stack value is result.

**Notes**:

* Implementation can be simplified if you require tokens to be **space-separated** (reduces tokenizer complexity).
* Precedence: ^ > \* / > + -. ^ is typically right-associative.

**Program: Infix to Postfix (space-separated tokens) & evaluate**

**File:** InfixToPostfix.java

// InfixToPostfix.java

import java.util.\*;

public class InfixToPostfix {

// Assumes tokens are space-separated: e.g. "3 + 4 \* 2 / ( 1 - 5 ) ^ 2 ^ 3"

public static String infixToPostfix(String expr) {

String[] tokens = expr.trim().split("\\s+");

StringBuilder output = new StringBuilder();

Deque<String> ops = new ArrayDeque<>();

for (String token : tokens) {

if (isNumber(token)) {

output.append(token).append(' ');

} else if (token.equals("(")) {

ops.push(token);

} else if (token.equals(")")) {

while (!ops.isEmpty() && !ops.peek().equals("(")) output.append(ops.pop()).append(' ');

if (ops.isEmpty()) throw new IllegalArgumentException("Mismatched parentheses");

ops.pop(); // remove "("

} else if (isOperator(token)) {

while (!ops.isEmpty() && isOperator(ops.peek())) {

String top = ops.peek();

if ((isLeftAssoc(token) && precedence(token) <= precedence(top)) ||

(!isLeftAssoc(token) && precedence(token) < precedence(top))) {

output.append(ops.pop()).append(' ');

} else break;

}

ops.push(token);

} else throw new IllegalArgumentException("Unknown token: " + token);

}

while (!ops.isEmpty()) {

String t = ops.pop();

if (t.equals("(") || t.equals(")")) throw new IllegalArgumentException("Mismatched parentheses");

output.append(t).append(' ');

}

return output.toString().trim();

}

public static double evaluatePostfix(String postfix) {

Deque<Double> st = new ArrayDeque<>();

if (postfix.trim().isEmpty()) return 0;

for (String token : postfix.split("\\s+")) {

if (isNumber(token)) st.push(Double.parseDouble(token));

else if (isOperator(token)) {

if (st.size() < 2) throw new IllegalArgumentException("Malformed postfix");

double b = st.pop();

double a = st.pop();

st.push(applyOp(a, b, token));

} else throw new IllegalArgumentException("Unknown token: " + token);

}

if (st.size() != 1) throw new IllegalArgumentException("Malformed postfix");

return st.pop();

}

private static boolean isNumber(String s) {

try { Double.parseDouble(s); return true; }

catch (NumberFormatException e) { return false; }

}

private static boolean isOperator(String s) { return "+-\*/^".contains(s) && s.length() == 1; }

private static int precedence(String op) {

switch (op) {

case "^": return 4;

case "\*": case "/": return 3;

case "+": case "-": return 2;

default: return 0;

}

}

private static boolean isLeftAssoc(String op) { return !op.equals("^"); }

private static double applyOp(double a, double b, String op) {

switch (op) {

case "+": return a + b;

case "-": return a - b;

case "\*": return a \* b;

case "/": return a / b;

case "^": return Math.pow(a, b);

default: throw new IllegalArgumentException("Unknown operator");

}

}

public static void main(String[] args) {

String infix = "3 + 4 \* 2 / ( 1 - 5 ) ^ 2 ^ 3";

String postfix = infixToPostfix(infix);

System.out.println("Infix : " + infix);

System.out.println("Postfix: " + postfix);

System.out.println("Value : " + evaluatePostfix(postfix));

String infix2 = "-2 + 3 \* ( 4 - 1 )"; // if you want unary minus, write it as "0 - 2 + ..."

// For simplicity this implementation assumes unary minus is handled in input (like "0 - 2")

}

}

**Program explanation (block):**

* infixToPostfix:
  + Splits the expression into tokens by spaces.
  + If token is number → append to output.
  + If token ( → push onto operator stack.
  + If token ) → pop operators to output until (.
  + If operator → apply precedence & associativity rules to pop appropriate operators to output, then push current operator.
  + At end pop remaining operators.
* evaluatePostfix:
  + Split postfix tokens; push numbers; for operator pop b then a, compute a op b, push result.
* Because tokenizer expects spaces, -2 as unary minus must be represented properly (e.g., 0 - 2), or you can extend tokenizer to detect unary minus (longer code).
* Complexity: conversion O(n), evaluation O(n). Uses O(n) extra space (stacks).

**4C Function call stack (visualizing recursion via prints)**

**Concept**

* Each function call gets a stack frame: parameters, local variables, return address.
* As recursion deepens frames stack up; returns pop frames in reverse order.

**Program: Recursion trace to demonstrate call stack**

**File:** RecursionTrace.java

// RecursionTrace.java

public class RecursionTrace {

public static int factorialTrace(int n) {

System.out.println("enter factorial(" + n + ")");

if (n == 0 || n == 1) {

System.out.println("return 1 from factorial(" + n + ")");

return 1;

}

int result = n \* factorialTrace(n - 1);

System.out.println("return " + result + " from factorial(" + n + ")");

return result;

}

public static void main(String[] args) {

int r = factorialTrace(4);

System.out.println("factorial(4) = " + r);

}

}

**Program explanation:**

* Each call prints when entering and before returning.
* Running factorialTrace(4) produces nested prints that represent how frames are pushed (enter) and popped (return).
* This makes it easy to visualize the call stack.

**5. Summary of complexities & practical tips**

* **Recursion**: clear for divide-and-conquer; watch stack depth and prefer iterative or tail-optimized alternatives in Java for deep recursions.
* **Stack ADT**: push/pop/peek = O(1). Implemented via arrays (amortized O(1) push) or linked lists (O(1) push/pop).
* **Parentheses matching**: O(n) time, O(n) auxiliary space.
* **Infix→Postfix & evaluation**: O(n) time, O(n) space. Use Shunting-yard for conversion.
* **When to use arrays vs linked list for stack**:
  + Use array-backed stack when you want better locality and expect fewer resizes.
  + Use linked-stack when memory resizing should be avoided and each push/pop must be guaranteed O(1) without amortization.

**Queue ADT — Concept**

**What is a Queue?**

* **Queue ADT** (Abstract Data Type): A linear data structure that follows the **FIFO principle** — *First In, First Out*.
* **Analogy:** Like a line at a ticket counter — first person in line is served first.

**FIFO Principle**

* **FIFO (First-In-First-Out):**  
  The element inserted earliest is the first to be removed.
* **Contrast with Stack:** Stack is LIFO (last-in-first-out).

**Basic Queue Operations**

1. **Enqueue(x):** Insert x at the rear (end).
2. **Dequeue():** Remove and return the element at the front.
3. **Front():** View the front element without removing.
4. **isEmpty():** Check if the queue is empty.
5. **isFull():** (For fixed-size arrays) Check if the queue is full.

**Time Complexity:**  
All operations should ideally be **O(1)**.

**2️⃣ Queue Implementation Using Arrays (Linear Queue)**

**Concept**

* Use an array arr[], two indices:
  + front: index of current front element
  + rear: index where next element will be inserted
* Problem: After many dequeues, space at the start of the array becomes unused (solution → circular queue).

**Program: Array-based Queue (Linear)**

// ArrayQueue.java

import java.util.NoSuchElementException;

public class ArrayQueue {

private int[] data;

private int front, rear, size;

public ArrayQueue(int capacity) {

data = new int[capacity];

front = 0;

rear = -1;

size = 0;

}

public void enqueue(int value) {

if (isFull()) throw new IllegalStateException("Queue is full");

rear++;

data[rear] = value;

size++;

}

public int dequeue() {

if (isEmpty()) throw new NoSuchElementException("Queue is empty");

int val = data[front];

front++;

size--;

return val;

}

public int front() {

if (isEmpty()) throw new NoSuchElementException("Queue is empty");

return data[front];

}

public boolean isEmpty() { return size == 0; }

public boolean isFull() { return size == data.length; }

public int size() { return size; }

public static void main(String[] args) {

ArrayQueue q = new ArrayQueue(5);

q.enqueue(10);

q.enqueue(20);

q.enqueue(30);

System.out.println("Front: " + q.front()); // 10

System.out.println("Dequeue: " + q.dequeue()); // 10

System.out.println("Dequeue: " + q.dequeue()); // 20

System.out.println("Size: " + q.size()); // 1

}

}

**Explanation:**

* enqueue increases rear, places value, increments size.
* dequeue removes from front and increments front.
* **Limitation:** Once rear reaches data.length-1, queue cannot accept more elements even if there’s free space in the beginning (solved by circular queue).

**3️⃣ Circular Queue (Array-based)**

**Concept**

* Uses modular arithmetic (% capacity) to wrap around when rear or front reaches the end of array.
* Ensures **space is reused** efficiently.

**Program: Circular Queue Implementation**

// CircularQueue.java

import java.util.NoSuchElementException;

public class CircularQueue {

private int[] data;

private int front, rear, size;

public CircularQueue(int capacity) {

data = new int[capacity];

front = 0;

rear = -1;

size = 0;

}

public void enqueue(int value) {

if (isFull()) throw new IllegalStateException("Queue is full");

rear = (rear + 1) % data.length;

data[rear] = value;

size++;

}

public int dequeue() {

if (isEmpty()) throw new NoSuchElementException("Queue is empty");

int val = data[front];

front = (front + 1) % data.length;

size--;

return val;

}

public int front() {

if (isEmpty()) throw new NoSuchElementException("Queue is empty");

return data[front];

}

public boolean isEmpty() { return size == 0; }

public boolean isFull() { return size == data.length; }

public int size() { return size; }

public static void main(String[] args) {

CircularQueue cq = new CircularQueue(3);

cq.enqueue(1); cq.enqueue(2); cq.enqueue(3);

System.out.println("Dequeue: " + cq.dequeue()); // 1

cq.enqueue(4); // reuses freed space

while (!cq.isEmpty()) System.out.println(cq.dequeue());

}

}

**Explanation:**

* rear and front use (index + 1) % capacity to wrap around.
* Space is fully utilized (no wasted slots).
* Time complexity remains **O(1)**.

**4️⃣ Linked List Implementation of Queue**

**Concept**

* Use a **singly linked list** with:
  + front pointing to first node
  + rear pointing to last node
* No fixed capacity — grows dynamically.

**Program: Linked List Queue**

// LinkedListQueue.java

import java.util.NoSuchElementException;

public class LinkedListQueue {

private static class Node {

int data;

Node next;

Node(int d) { data = d; }

}

private Node front, rear;

private int size;

public void enqueue(int value) {

Node newNode = new Node(value);

if (rear != null) rear.next = newNode;

rear = newNode;

if (front == null) front = newNode;

size++;

}

public int dequeue() {

if (isEmpty()) throw new NoSuchElementException("Queue is empty");

int val = front.data;

front = front.next;

if (front == null) rear = null; // queue became empty

size--;

return val;

}

public int front() {

if (isEmpty()) throw new NoSuchElementException("Queue is empty");

return front.data;

}

public boolean isEmpty() { return front == null; }

public int size() { return size; }

public static void main(String[] args) {

LinkedListQueue q = new LinkedListQueue();

q.enqueue(100); q.enqueue(200);

System.out.println("Front: " + q.front()); // 100

System.out.println("Dequeue: " + q.dequeue()); // 100

System.out.println("Dequeue: " + q.dequeue()); // 200

}

}

**Explanation:**

* enqueue: Adds new node at rear in O(1).
* dequeue: Removes node from front in O(1).
* Efficient and no wasted memory, but extra memory per node (next pointer).

**5️⃣ Priority Queue (Heap-based)**

**Concept**

* A **Priority Queue** removes elements based on priority rather than order of insertion.
* Implemented internally using **heap data structure** (binary heap).
* **Java:** Provides PriorityQueue<E> class in java.util.

**Program: Priority Queue Example**

// PriorityQueueExample.java

import java.util.PriorityQueue;

public class PriorityQueueExample {

public static void main(String[] args) {

PriorityQueue<Integer> pq = new PriorityQueue<>(); // Min-heap by default

pq.offer(30);

pq.offer(10);

pq.offer(20);

System.out.println("Front (smallest): " + pq.peek()); // 10

while (!pq.isEmpty()) {

System.out.println("Dequeue: " + pq.poll());

}

}

}

**Explanation:**

* offer = enqueue, poll = dequeue (removes smallest element).
* By default, PriorityQueue is a **min-heap** (lowest value has highest priority).
* For max-heap, use custom comparator:  
  PriorityQueue<Integer> pq = new PriorityQueue<>(Collections.reverseOrder());

**6️⃣ Real-Time Use Cases**

| **Queue Type** | **Real-World Example** |
| --- | --- |
| **Simple Queue** (FIFO) | Printer spooler, task scheduling |
| **Circular Queue** | OS process scheduling (Round Robin) |
| **Linked List Queue** | Messaging systems (no fixed capacity) |
| **Priority Queue** | Dijkstra’s shortest path, CPU job scheduling, event simulation |