ALLAMA IQBAL OPEN UNIVERSITY



AIOU LIBRARY
aioulibrary.com

Name:

Registration No.:

Course Code: 3581

Program: BS

Semester: Autumn 2026

ALLAMA IQBAL OPEN UNIVERSITY, ISLAMABAD

(Department of Computer Science)

Course: Data Structure and Application (3581) • Level: BS

Semester: Autumn 2026

Assignment No. 1

Total Marks: 100 • Pass Marks: 50

SOLVED ASSIGNMENT GUIDE — For Study & Reference

Note: This guide is prepared for study and reference purposes only. Students are advised to read the relevant parts of the course book, understand the concepts, try the programming examples themselves on their own computer, and then write the assignment in their own words. Copying this guide word for word is not recommended. No units are printed in the question booklet for this course, so none are shown on this title page.

Question 1: Array versus Linked List — Memory Allocation; Inserting a Node at the Beginning of a Singly Linked List (20 Marks)

Question as printed: What is the difference between an Array and a Linked List in terms of memory allocation? Write a program in C++ or Python to insert a new node at the beginning of a Singly Linked List.

Introduction

Almost every program works with a collection of values: a list of students, a set of marks, a queue of jobs or a collection of records read from a file. The two most elementary ways of

holding such a collection in memory are the array and the linked list, and the choice between them is one of the first real design decisions in data structures. Superficially they do the same job — both store a sequence of elements that can be traversed in order — but they obtain, organise and release memory in completely different ways, and those differences decide how fast each operation is, how much memory is wasted, and how easily the collection can grow. This answer compares the two structures strictly from the point of view asked in the question, namely memory allocation, supported by the related consequences for access, insertion and deletion, and then gives a complete, tested Python program that inserts a new node at the beginning of a singly linked list, the simplest and most instructive linked-list operation.

What an Array Is, and How It Allocates Memory

An array is a collection of elements of the same type stored in one continuous, unbroken block of memory. When a program declares an array of, say, 100 integers, the system must find 100 consecutive locations and reserve the whole block at once; this is called contiguous allocation. Because the elements sit side by side at equal intervals, the address of any element can be calculated directly from the address of the first element and the index: address of element i equals the base address plus i multiplied by the size of one element. That single arithmetic step is why arrays give O(1), constant-time, random access — the tenth, thousandth or millionth element is reached just as quickly as the first. The price of this elegance is rigidity. The size is normally fixed at the time of creation, so the programmer must predict the maximum number of elements in advance. Predicting too generously wastes memory that no other part of the program can use; predicting too little causes overflow, and enlarging the collection means allocating a completely new, larger block, copying every element into it and releasing the old

block, which is an O(n) operation. In languages such as C and C++ an array created inside a function is usually allocated statically or on the stack, while in Python a built-in list is a dynamic array that hides this copying from the programmer but still performs it behind the scenes whenever it grows beyond its current capacity.

What a Linked List Is, and How It Allocates Memory

A linked list takes the opposite approach. Its elements are held in separate objects called nodes, and each node contains two parts: the data itself and a pointer (reference) to the next node in the sequence. The nodes are allocated one by one, dynamically, at run time, exactly when they are needed, and they may be scattered anywhere in memory — wherever the allocator happens to find a free place. This is non-contiguous, dynamic allocation. The sequence is preserved not by physical nearness but by the chain of pointers: a special pointer called head points to the first node, each node points to its successor, and the last node points to None (NULL), marking the end. Because no single large block is required, a linked list can grow and shrink element by element without ever copying the existing data, and it never wastes space on unused capacity. The costs are equally real. Every node carries the extra memory of its pointer, so a linked list uses more memory per element than an array holding the same data. There is no address arithmetic, so there is no random access: reaching the fifth element requires starting at head and following four pointers, an O(n) sequential search. Finally, scattered nodes make poor use of the processor cache, which is designed to fetch neighbouring memory quickly, so in practice linked-list traversal is often slower than array traversal even when the theoretical operation counts look similar.

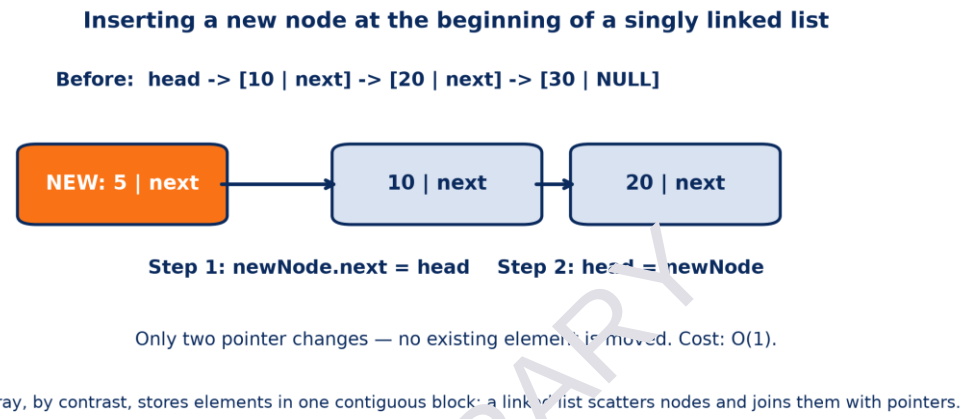

Figure 1: Insertion at the beginning of a singly linked list needs only two pointer changes, because linked-list nodes are allocated separately and joined by pointers rather than stored in one block.

Table 1: Array and linked list compared on memory allocation and its consequences

Aspect	Array	Linked List
Allocation style	Contiguous: one unbroken block for all elements	Non-contiguous: each node allocated separately, wherever space is available
When memory is obtained	Usually all at once, at creation (static or stack allocation; dynamic arrays reserve spare capacity)	Dynamically, one node at a time, exactly when an element is added
Size	Fixed in basic form; growing requires a new block and copying	Flexible; grows and shrinks node by node with no copying
Extra memory per element	None; only the data is stored	One pointer per node (two in a doubly linked list)

Wastage	Unused reserved capacity is wasted; overflow is possible	No unused capacity, but pointer overhead is always paid
Access to element i	O(1) by address calculation from the base address	O(n) by following pointers from head
Insert / delete in the middle	O(n): later elements must be shifted	O(1) once the position is reached: only pointers change
Cache behaviour	Excellent: neighbouring elements are fetched together	Poor: nodes are scattered in memory

A Balanced Judgement: When Each Structure Is the Right Choice

Neither structure is universally better; each is better at the work the other finds expensive. An array is the right choice when the number of elements is known or changes rarely, when fast access by index is the dominant operation — as in mathematical vectors, image pixels, tables of marks or the array-based queues and hash tables used later in this course — and when memory must be used economically. A linked list is the right choice when insertions and deletions are frequent, especially at the beginning, when the final size cannot be predicted, or when other structures are built on top of it: the adjacency lists of Assignment No. 2, Question 4, the chaining used in hash tables, and the dynamic stacks and queues of operating systems are all linked structures underneath. Real systems frequently combine the two: Python's list is a dynamic array, a database index keeps sorted keys in array-like pages linked together, and the circular queue of Question 3 is deliberately built on an array precisely because a queue needs predictable, index-based access. Understanding the memory-allocation trade-off — contiguity

and calculable addresses in exchange for rigidity, versus flexibility and pointer overhead in exchange for sequential access — is therefore the foundation on which the rest of this course is built.

Program: Inserting a New Node at the Beginning of a Singly Linked List

The program below is written in Python, which the question expressly allows as an alternative to C++. It defines a Node class, builds an initial list containing 10, 20 and 30, displays it, inserts a new node with value 5 at the beginning by performing exactly the two pointer steps shown in Figure 1 — point the new node at the old head, then make head point at the new node — and displays the list again. The logic is identical in C++: a new node is created with new, its next field is set to head, and head is updated. The operation costs O(1) time and creates exactly one new node of memory, which illustrates the dynamic-allocation argument of this answer in working code.

```
class Node:
    def __init__(self, data):
        self.data = data      # the value stored in this node
        self.next = None      # pointer to the next node


def insert_at_beginning(head, new_data):
    new_node = Node(new_data)  # Step 0: allocate one new node
    new_node.next = head       # Step 1: new node points to old first
node
    head = new_node            # Step 2: head now points to the new
node
```

```
    return head


def display(head):
    current = head
    while current is not None:
        print(current.data, end=" -> ")
        current = current.next
    print("None")


# Build an initial list  10 -> 20 -> 30 -> None
head = None
for value in [30, 20, 10]:
    head = insert_at_beginning(head, value)
print("List before insertion:")
display(head)


# Insert 5 at the beginning
head = insert_at_beginning(head, 5)
print("List after inserting 5 at the beginning:")
display(head)
```

Output of the program: List before insertion: 10 -> 20 -> 30 -> None. List after inserting 5 at the beginning: 5 -> 10 -> 20 -> 30 -> None. Three points in the working deserve emphasis. First, no existing node was moved, copied or even visited, which is why insertion at the beginning is a constant-time operation in a linked list, whereas inserting at the beginning of an array would

require shifting every element one place to the right. Second, the order of the two steps matters: if head were updated before the new node's next pointer was set, the address of the old list would be lost and the whole list would become unreachable. Third, the empty-list case needs no special code: if head is None, the new node's next simply becomes None and it correctly becomes the only node, which is why the building loop above could start from an empty list and use the same function for every element.

Conclusion

In terms of memory allocation, an array reserves one contiguous block in advance, pays nothing extra per element and gains constant-time access by address calculation, but risks wasted capacity and expensive resizing; a linked list allocates each node dynamically and separately, pays a pointer for every element and can reach elements only by following the chain, but grows and shrinks freely with no copying and no waste of unused capacity. The insert-at-beginning program demonstrates the practical consequence of that design: a new element is added with one fresh allocation and two pointer assignments, in O(1) time, regardless of how long the list already is — the clearest possible illustration of why linked allocation exists alongside the older, simpler array.

Question 2: Stack Operations (Push and Pop); Checking Balanced Parentheses with a Stack (20 Marks)

Question as printed: Describe the basic operations of a Stack (Push and Pop). Write a function/program that uses a stack to check if a given string of parentheses (e.g., "{[()]}") is balanced.

Introduction

A stack is the most disciplined of all data structures: elements may be added and removed at one end only, called the top, so that the last element inserted is always the first element removed. This rule, known as Last In, First Out or LIFO, sounds restrictive, yet it matches a remarkable number of real problems exactly — undo commands, function calls, backtracking, expression evaluation and, as this question shows, checking that brackets in a program or expression are correctly matched. This answer first describes the structure and its basic operations, Push and Pop, together with the supporting operations and their costs; it then explains why bracket matching is naturally a stack problem, gives a complete Python program that checks whether a string of parentheses is balanced, and traces the given example by hand so that the program's behaviour can be verified without running it.

The Stack and Its Basic Operations

It helps to picture a stack of plates in a canteen: a clean plate is placed on top of the pile, and the next customer takes the plate from the top. Nobody removes a plate from the middle, because that would disturb everything above it. A data-structure stack behaves in exactly this

way. Formally it is an abstract data type defined not by how it is stored but by the operations it supports and the LIFO rule they obey. A stack may be implemented with an array and an index marking the top, or with a linked list whose head acts as the top; in Python the built-in list, used only through append and pop from the end, is a convenient and efficient stack. Whatever the implementation, the interface and the costs are the same, and every stack operation is O(1) because work is only ever done at the top.

Push — Adding an Element at the Top

Push inserts a new element at the top of the stack. In an array implementation the top index is increased by one and the value is stored at that position; in a linked-list implementation a new node is created and linked in at the head. Before pushing onto a fixed-size array stack, a well-written program checks for overflow — the condition in which the stack is already full and has no room for another element. After a successful push, the newly inserted element becomes the current top and will be the first element returned by a pop. Push is how a program records work still to be finished: an opening bracket waiting for its closing bracket, a function call waiting to return, or an action waiting to be undone.

Pop — Removing and Returning the Top Element

Pop removes the top element from the stack and returns its value to the caller. In an array implementation the value at the top index is taken and the index is decreased by one; in a linked list the head node is unlinked and its memory released. Before popping, a program must check for underflow — trying to pop from an empty stack — because there is no top element to return, and treating that situation as an error (or as evidence of unbalanced input, as in the

program below) is far safer than returning a meaningless value. Pop is the mirror image of push: repeated pops return the elements in exactly the reverse of the order in which they were pushed, and it is this reversal property that makes the stack the right tool for matching, undoing and backtracking.

Supporting Operations: Peek, isEmpty and Size

Three supporting operations complete the usual interface. Peek (also called top) returns the value of the top element without removing it, allowing a program to inspect what it is about to deal with — the bracket checker below conceptually peeks when it compares the stack top with a closing bracket. isEmpty reports whether the stack contains no elements, which is both the underflow guard for pop and, in the bracket problem, the final test of success. Size reports how many elements are currently stored. None of these changes the LIFO rule; they simply make the two basic operations safe and usable. Table 1 summarises the full set.

Table 1: The operations of a stack

Operation	What it does	Cost	Error / special condition
Push(x)	Places x on the top of the stack	O(1)	Overflow if a fixed-size stack is full
Pop()	Removes and returns the top element	O(1)	Underflow if the stack is empty
Peek() / Top()	Returns the top element without removing it	O(1)	Undefined / error if the stack is empty

isEmpty()	True if no elements are stored	O(1)	—
Size()	Number of elements currently stored	O(1)	—

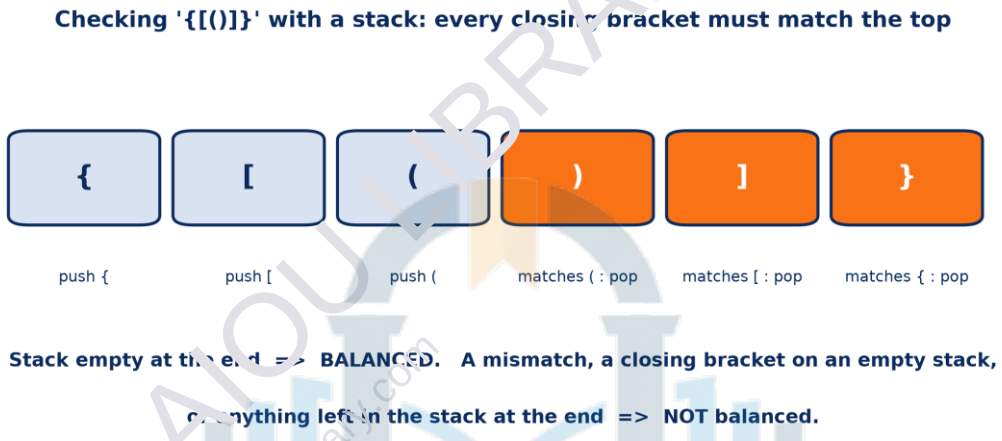

Figure 1: Tracing '{[()]}' through a stack. Opening brackets are pushed; each closing bracket must match and pop the top. An empty stack at the end means the string is balanced.

Why Bracket Matching Is a Stack Problem

A string of brackets is balanced when every opening bracket — (, [or { — is closed by a bracket of the same type, and the pairs are correctly nested: the bracket opened most recently must be closed first. In the string {[()]}, for example, the { opened first is closed last, while the (opened just before its) is closed immediately. Correct nesting is precisely the LIFO pattern: at any closing bracket, the only opening bracket it is allowed to match is the most recent still-unmatched one, and that is exactly the element sitting on the top of a stack that has been fed the opening brackets in order. A simple counter of opening and closing brackets would not be

enough, because it would accept a wrongly ordered string such as {[(])}, in which the counts are equal but the nesting is broken. The algorithm therefore reads the string from left to right: push every opening bracket; on meeting a closing bracket, check that the stack is not empty and that popping it gives the matching opening type; ignore any characters that are not brackets, if the application allows them. If every closing bracket matches and the stack is empty when the string ends, the string is balanced. Any mismatch, any closing bracket arriving at an empty stack, or any opening bracket left in the stack at the end proves the string unbalanced.

Program: Checking Balanced Parentheses with a Stack

The Python function below implements the algorithm exactly as described. A dictionary records which opening bracket each closing bracket requires, so the matching test is a single comparison. The function returns True or False, and the short driver code tests it on the example from the question and on deliberately unbalanced strings, so its behaviour can be checked against the hand trace that follows.

```
def is_balanced(expression):
    stack = []                          # empty stack (Python list)
    matching = {')': '(', ']': '[', '}': '{'}

    for char in expression:
        if char in "([{":
            stack.append(char)          # Push opening bracket
        elif char in ")]}":
            if not stack:               # closing bracket, nothing
to match
```

```
                return False
            top = stack.pop()                # Pop the most recent
opening bracket
            if top != matching[char]:
                return False                 # wrong type: not properly
nested
    return len(stack) == 0                   # balanced only if nothing
is left

tests = ["{[()]}", "{[(])}", "(())))", "(()", ")(", "{[()]}()"]
for t in tests:
    print(t, "->", "Balanced" if is_balanced(t) else "Not Balanced")
```

Output of the program: {[()]} -> Balanced; {[(])} -> Not Balanced; ((())) -> Balanced; (() -> Not Balanced;)(-> Not Balanced; {[()]}() -> Balanced. Now trace the required example, {[()]}, step by step. Reading { pushes it, giving stack [{]. Reading [pushes it, giving [{, []. Reading (pushes it, giving [{, [, (]. Reading) requires (, and popping gives (— a match, leaving [{, []. Reading] requires [, and popping gives [— a match, leaving [{]. Reading } requires {, and popping gives { — a match, leaving the stack empty. The string has ended and the stack is empty, so the function returns True: the expression is balanced, exactly as the eye confirms from its perfectly nested structure. Contrast {[(])}: after pushing {, [and (, the character] demands [but the pop returns (, so the function correctly returns False even though the string contains equal numbers of opening and closing brackets — the test that defeats a simple counter. The algorithm

examines each character once and stores at most one stack entry per opening bracket, so it runs in O(n) time and O(n) worst-case extra memory for a string of length n.

Where Stacks Are Used Beyond This Question

The same push-pop discipline appears throughout computing, which is why examiners return to it so often. Every running program keeps a call stack: calling a function pushes an activation record holding its parameters, local variables and return address, and returning pops it — this is the mechanism that also makes the recursion of Question 4 work. Compilers and calculators use stacks to evaluate expressions, convert infix notation such as (A + B) * C into postfix form, and check the syntax of programs, of which bracket matching is the simplest visible part. Editors, browsers and drawing programs implement undo by pushing each action onto a stack and popping it when the user steps backwards, and web browsers keep back and forward histories on the same principle. Depth-first search, maze solving and other backtracking algorithms push each choice as it is made and pop it when a dead end is reached. In each case the problem shares the structure of the bracket question: something is opened, begun or chosen, and must later be closed, finished or undone in exactly the reverse order.

Conclusion

A stack is a LIFO structure in which Push adds an element at the top and Pop removes and returns that top element, supported by Peek, isEmpty and Size, all in constant time, with overflow and underflow as the conditions a careful programmer guards against. Because correctly nested brackets must be closed in the reverse order of their opening, a stack settles the balanced-parentheses question in a single left-to-right pass: push opening brackets, match

and pop on every closing bracket, and accept only if nothing remains. The program and trace above show the method accepting {[()]} and rejecting strings whose counts agree but whose nesting does not — the distinction that makes the stack, and not a mere counter, the right data structure for the task.

Question 3: The Circular Queue — Overcoming the Limitations of a Linear Queue; The Enqueue Operation (20 Marks)

Question as printed: What is a Circular Queue? Explain how it overcomes the limitations of a standard Linear Queue. Write a program to implement the enqueue operation for a Circular Queue using an array.

Introduction

A queue is the natural counterpart of the stack of Question 2: elements are inserted at one end, called the rear, and removed from the other end, called the front, so that the first element inserted is the first element removed — First In, First Out, or FIFO. Queues model fairness in real life and in computing alike: a line at a ticket window, print jobs waiting for a printer, keystrokes waiting to be processed, and patients waiting to be seen. The straightforward way to implement a queue is with an array and two indices, front and rear, but that simple linear queue contains a frustrating defect: it can report itself full while much of its array is empty and unusable. The circular queue was invented to cure exactly that defect, by treating the array as if its ends were joined into a ring. This answer explains the linear queue and its limitation with a concrete example, defines the circular queue and shows how its modular arithmetic removes

the defect, and then gives a complete Python program for the enqueue operation, with the matching dequeue and display operations included so that the structure can be seen working as a whole.

The Linear Queue and Its Limitation

In a linear (simple) queue implemented on an array of size n, front points to the oldest element and rear to the newest. Enqueue advances rear and stores the new value; dequeue takes the value at front and advances front. Both indices therefore march steadily towards the high end of the array, and neither ever moves backwards. The defect follows immediately. Suppose an array of size 5 receives five elements, so rear reaches index 4, and three elements are then removed, so front advances to index 3. Two elements remain, and three array positions — indices 0, 1 and 2 — are empty. Yet the next enqueue must be refused, because rear has already reached the last index and the usual full test, rear == size - 1, declares the queue full. Memory that has been paid for and freed by dequeues cannot be reused. The only remedies inside the linear design are unattractive: shifting all remaining elements back to the start on every dequeue, which makes dequeue an O(n) operation, or abandoning the wasted space. This false-full condition — sometimes called the limitation or drawback of the linear queue — is the whole reason the circular design exists.

What a Circular Queue Is

A circular queue is a queue implemented on an array whose last position is treated as if it were followed immediately by the first position, so the indices form a circle rather than a line. The device that creates the circle is the remainder operator: whenever an index is advanced, it is

advanced as (index + 1) % size, where % gives the remainder on division. With size 5, advancing from index 4 gives (4 + 1) % 5 = 0, so rear wraps around to the beginning of the array and can reuse exactly the positions that dequeue has freed. Front wraps in the same way when elements are removed. The queue is genuinely full only when every position is occupied — tested, in the implementation used below, by checking whether advancing rear once more would collide with front, that is, (rear + 1) % size == front — and genuinely empty when front is in its initial, unused state (represented here by -1, reset whenever the last element is removed). Nothing else about the queue changes: insertion still happens at the rear, removal still happens at the front, FIFO order is preserved exactly, and both operations remain O(1). The circular queue is also called a ring buffer, a name that reveals how widely it is used wherever a fixed-size, continuously reused store is needed.

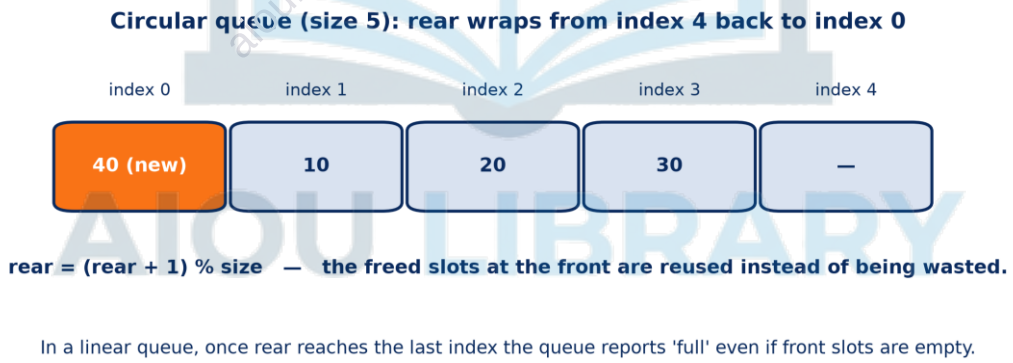

Figure 1: In a circular queue the rear pointer wraps from the last index back to index 0, so positions freed at the front are filled again instead of being wasted.

Table 1: Linear queue and circular queue compared

Aspect	Linear Queue	Circular Queue

Index movement	front and rear only move forward	Indices advance with (index + 1) % size and wrap around
Use of freed positions	Positions before front can never be reused	Freed positions are reused automatically
False-full condition	Reports full when rear reaches the end, even with empty slots	Reports full only when all positions are occupied
Full test	rear == size - 1	(rear + 1) % size == front
Empty test	front == -1 or front > rear	front == -1 (in this implementation)
Enqueue / dequeue cost	O(1) unless elements are shifted to reclaim space	O(1) always, with no shifting
Memory efficiency	Poor: part of the array becomes permanently unusable	Good: the whole array stays in continuous use

How the Circular Queue Overcomes the Limitation

The cure can be stated in one sentence: because indices are computed modulo the size, the space a dequeue releases at the beginning of the array becomes the space the next enqueues consume, so the array is used as a closed loop instead of a one-way track. Returning to the earlier example, the circular queue holding two elements at indices 3 and 4 happily accepts the next element at index 0, the next at index 1, and the next at index 2, and only then, with all five positions genuinely occupied, does it report full. No element is ever shifted, no copy is made, and the amount of usable storage rises from whatever happens to remain after rear to the full capacity of the array at all times. This matters in precisely the long-running, continuous

workloads queues serve: an operating system's input buffer, a network device's packet buffer, a printer spooler or a call-centre system runs for hours or days, processing far more items in total than its buffer could ever hold at once. A linear queue in such a role would exhaust itself within the first minutes of operation even though it was nearly empty; a circular queue in the same role runs indefinitely, which is why ring buffers are a standard component of operating systems, device drivers, audio and video streaming, and producer-consumer systems generally. The trade-off is modest: the index arithmetic is slightly less obvious than in a linear queue, and one standard convention — including the implementation below — effectively distinguishes full from empty by the position test rather than by a separate counter, a small price examined carefully in programming exercises on this topic. An alternative implementation keeps an explicit count of elements, which makes the full and empty tests trivially count == size and count == 0, at the cost of maintaining one more variable correctly.

Program: Enqueue (with Dequeue and Display) for a Circular Queue

The program below implements a circular queue of size 5 in Python, using a fixed list to play the role of the array. The enqueue function, which is the operation the question asks for, is written out in full and commented step by step: reject the new value if the queue is full, handle the very first insertion by setting front to 0, advance rear with modular arithmetic, and store the value. Dequeue and display are included because a queue operation can only be demonstrated convincingly by moving elements both in and out and showing the wrap-around actually happening: the driver code fills the queue, removes two elements, and then enqueues two more, which are placed in the wrapped positions at the start of the array.

```
SIZE = 5
queue = [None] * SIZE
front = -1
rear = -1


def enqueue(value):
    global front, rear
    # Full test: the next rear position must not collide with front
    if (rear + 1) % SIZE == front:
        print("Queue is FULL. Cannot enqueue", value)
        return
    if front == -1:          # first element being inserted
        front = 0
    rear = (rear + 1) % SIZE  # advance rear, wrapping around
    queue[rear] = value
    print("Enqueued:", value)


def dequeue():
    global front, rear
    if front == -1:
        print("Queue is EMPTY. Nothing to dequeue")
        return None
    value = queue[front]
    queue[front] = None
    if front == rear:         # that was the last element
        front = -1
        rear = -1
```

```
        else:
            front = (front + 1) % SIZE  # advance front, wrapping around
        print("Dequeued:", value)
        return value


def display():
    print("Queue array:", queue, "| front =", front, "| rear =", rear)


enqueue(10); enqueue(20); enqueue(30); enqueue(40); enqueue(50)
display()
dequeue(); dequeue()
display()
enqueue(60); enqueue(70)   # these wrap around to indices 0 and 1
display()
enqueue(80)                # queue is genuinely full now
```

Reading the output confirms the argument of this answer. After the first five enqueues the array is [10, 20, 30, 40, 50] with front 0 and rear 4, and it is truly full. Dequeuing twice returns 10 and 20 and moves front to 2, freeing indices 0 and 1 — the exact situation in which a linear queue of the same history, had rear stayed at the end, would have been crippled once further enqueues pushed it to its limit. The circular enqueue now places 60 at index 0, because rear advances as (4 + 1) % 5 = 0, and places 70 at index 1, giving the array [60, 70, 30, 40, 50] with front 2 and rear 1; read in queue order from front, wrapping around, the contents are 30, 40, 50, 60, 70, in perfect FIFO order. The final attempt to enqueue 80 is correctly refused, because

(1 + 1) % 5 == 2 == front: every position is genuinely occupied. Students should note that display through the raw array can look disordered after wrapping, and that correct logical order is always recovered by starting at front and advancing modulo the size until rear is reached.

Conclusion

A circular queue is a FIFO queue whose array indices advance modulo the size, joining the end of the array back to its beginning in a ring. It overcomes the defining limitation of the linear queue — the false-full condition in which positions freed by dequeue could never be reused — by allowing rear and front to wrap around and refill those positions, so full means genuinely full and both enqueue and dequeue stay at O(1) with no shifting. The enqueue program above, with its collision-based full test and its (rear + 1) % size advance, is the heart of the structure, and the demonstration trace shows the rescued positions 0 and 1 accepting new elements exactly as the theory predicts.

Question 4: Recursion — Base Case and Recursive Case; A Recursive Factorial Program (20 Marks)

Question as printed: Explain the concept of Recursion and the difference between the Base Case and the Recursive Case. Write a recursive program to calculate the factorial of a number entered by the user.

Introduction

Recursion is the technique in which a function solves a problem by calling itself on a smaller version of the same problem. It is at first sight a strange idea — a function defined in terms of

itself sounds like a dictionary that defines a word by using the word — yet it is one of the most powerful and elegant ideas in computing, because a great many problems are themselves self-similar: a factorial is a number multiplied by a slightly smaller factorial, a folder contains smaller folders, a tree is a node joined to smaller trees, and a search of a sorted list is a search of half a list. Wherever a problem naturally breaks into smaller copies of itself, a recursive solution mirrors that structure so directly that it is often shorter and clearer than any loop-based alternative. This answer explains the concept and the machinery underneath it, distinguishes carefully between the base case and the recursive case — the two components every correct recursive function must contain — discusses advantages, costs and common errors, and finishes with a complete, tested Python program that reads a number from the user and calculates its factorial recursively.

The Concept of Recursion

A recursive function has the following shape. Presented with a problem, it first asks whether the problem is already so small that it can be answered directly, without any further subdivision; if so, it returns that direct answer. Otherwise it reduces the problem to one or more strictly smaller instances of the same problem, solves them by calling itself, and combines the partial results into the answer for the original problem. The calls form a winding phase, in which each call pauses, half-finished, while its smaller call runs, followed by an unwinding phase, in which the smallest call returns first and each waiting caller resumes, completes its own small piece of work and returns in turn. The machinery that makes this possible is the call stack met in Question 2: every call pushes an activation record containing its own copy of the parameters and local variables, so ten simultaneous calls to the same function are ten

independent records, each minding its own value, and each pop delivers a result back to exactly the caller that was waiting for it. Nothing in the function text is shared except the code itself. It is worth stressing that recursion is a style of expressing a solution, not a different kind of computation: every recursive algorithm has an iterative equivalent using an explicit loop (and, where necessary, an explicit stack), and every loop can be rewritten recursively. The choice between them is a matter of clarity, cost and the natural shape of the problem.

The Base Case

The base case (also called the terminating case, stopping case or anchor) is the smallest, simplest instance of the problem, answered directly without a recursive call. For factorial, the base case is that 0! and 1! both equal 1 by definition, so the function simply returns 1. For searching a list it might be the empty list, for traversing a tree it is the empty tree or a leaf, and for a countdown it is reaching zero. The base case has two duties, and both are essential. First, it supplies the concrete values from which all larger answers are built during unwinding: the result of the factorial program is assembled entirely from the single 1 returned at the bottom, multiplied successively by 2, 3, 4 and so on as the calls return. Second, it guarantees termination. Because every recursive call works on a strictly smaller problem, the calls must eventually reach a size at which the base case applies, and at that moment the winding stops and unwinding begins. A recursive function with no base case, or with a base case its calls can never reach, calls itself forever — in practice until the call stack overflows and the program crashes with an error such as Python's RecursionError. Choosing the base case therefore comes first in recursive design: the programmer asks, before anything else, what the smallest meaningful problem is and what its answer is by definition.

The Recursive Case

The recursive case (also called the general case) is the part of the function that handles every problem too large for the base case. It performs three duties: it reduces the problem, moving it strictly closer to the base case; it calls the function itself on the reduced problem; and it combines the returned result with the part of the work belonging to the present call. For factorial the whole recursive case is contained in one expression, return n * factorial(n - 1): the problem n is reduced to n - 1, the recursive call solves that smaller problem completely, and multiplication by n combines its result into the answer for n. The strict reduction is not a detail but the logical heart of the method. If a call for n sometimes called the function for n again, or for n + 1, the calls would drift away from the base case instead of towards it, and termination would be lost. A useful discipline when writing or examining a recursive function is therefore to check three things in order: is there a base case, does every recursive call apply to a strictly smaller input, and is the returned result combined correctly? A function satisfying all three is correct by the same reasoning as mathematical induction, of which recursion is the computational mirror — the base case is the induction's starting point and the recursive case is its step from one size to the next.

Table 1: Base case and recursive case compared

Aspect	Base Case	Recursive Case
Purpose	Answers the smallest problem directly and stops the recursion	Reduces a larger problem and builds its answer from a smaller answer
Recursive call	None — the function returns	Yes — the function calls itself

	immediately	
Typical example (factorial)	if n == 0 or n == 1: return 1	return n * factorial(n - 1)
If missing or wrong	Infinite recursion and stack overflow / RecursionError	The function solves only the trivial problem and nothing larger
Analogy (mathematical induction)	The starting point: the statement checked for the smallest value	The inductive step: assuming size n - 1, proving size n
Design question it answers	What is the smallest problem, and what is its answer by definition?	How is this problem one step larger than a problem already solvable?

Advantages, Costs and Common Errors

Used where it fits, recursion offers three clear advantages. It produces code that closely matches the definition of the problem — the factorial, tree traversal and binary search programs in this assignment are each only a few lines, and read almost like the mathematics they implement. It divides a hard problem into manageable pieces without the bookkeeping variables an equivalent loop would need, because the call stack does the bookkeeping automatically. And it is the natural, sometimes the only comfortable, tool for inherently recursive data such as trees, graphs and nested file systems. The costs must be stated just as honestly. Every call consumes stack memory and call-and-return time, so a deeply recursive solution can be slower and more memory-hungry than a loop; Python, like many languages,

enforces a recursion limit (by default about 1,000 calls) precisely to stop runaway recursion from corrupting the program. Common beginner errors follow directly from the theory: omitting the base case, writing a recursive call that does not shrink the problem, shrinking towards the wrong base case, and using recursion for huge depths — computing the 5,000th term of a sequence naively — where an iterative method is safer. A further classic inefficiency is repeated work: a naive recursive Fibonacci function recomputes the same smaller values exponentially many times, a defect cured by storing results (memoisation) or by iterating. None of this argues against recursion; it argues for choosing it where the problem is self-similar and the depth is modest — conditions the factorial problem satisfies perfectly.

Program: Recursive Factorial of a Number Entered by the User

The program below reads a non-negative integer from the user, exactly as the question requires, validates it, and passes it to a recursive function whose two branches are the base case and the recursive case discussed above. A short trace for a small input follows the code, because being able to follow the winding and unwinding by hand is the real test of understanding recursion.

```
def factorial(n):
    # Base case: 0! = 1 and 1! = 1, by definition
    if n == 0 or n == 1:
        return 1
    # Recursive case: n! = n x (n - 1)!
    else:
        return n * factorial(n - 1)
```

```
# Input from the user
number = int(input("Enter a non-negative integer: "))


if number < 0:
    print("Factorial is not defined for negative numbers.")
else:
    result = factorial(number)
    print("Factorial of", number, "is", result)
```

Sample run: the user enters 5, and the program prints 'Factorial of 5 is 120'. Trace the winding phase first. factorial(5) cannot finish, because it needs factorial(4); it pauses with its multiplication by 5 pending. factorial(4) likewise needs factorial(3), which needs factorial(2), which needs factorial(1). Now the base case is reached: factorial(1) returns 1 immediately, with no further call. Unwinding begins. factorial(2) resumes and returns 2 x 1 = 2. factorial(3) returns 3 x 2 = 6. factorial(4) returns 4 x 6 = 24. factorial(5) finally returns 5 x 24 = 120, and the stack is empty again. Notice how the multiplications happen on the way back up, not on the way down, and in increasing order of size — a point students frequently reverse when tracing by hand. Checking other values confirms the program: factorial(0) returns 1 at once through the base case, agreeing with the mathematical convention 0! = 1; factorial(6) returns 720; and a negative input is rejected with a clear message instead of recursing downwards forever, which is itself an application of the termination lesson — the validation ensures the recursive case is only ever entered with values that genuinely approach the base case. The function's time cost is O(n), since each value from 1 to n is multiplied once, and its extra memory cost is O(n), since n

activation records can be on the call stack at the deepest moment — the standard, acceptable price of the recursive form.

Conclusion

Recursion solves a problem through self-similar smaller problems, relying on the call stack to keep each call's data separate while winding down to a stopping point and unwinding with the assembled answer. The base case answers the smallest problem directly and guarantees termination; the recursive case reduces every larger problem strictly towards that base case, calls the function on the reduction, and combines the result. The factorial program shows the partnership in its purest form — one line for the base case, one line for the recursive case, and a trace in which 5! = 120 emerges entirely from the single 1 returned at the bottom — while the discussion of costs and errors records the judgement a programmer needs: recursion where the problem's own structure is recursive and the depth is safe, iteration where it is not.

Question 5: The Binary Search Tree (BST); Insertion and In-Order Traversal (20 Marks)

Question as printed: Define a Binary Search Tree (BST). Write a program to insert three elements (e.g., 15, 10, and 20) into an empty BST and then write a function to perform an In-Order traversal to display the elements.

Introduction

Searching is the operation computing performs more than any other, and the history of data structures is largely a search for ways of searching faster than examining every element in turn.

A sorted array searched by the binary search of Assignment No. 2 is one answer, but it pays for its speed when elements are inserted or deleted, because half the array must shift to keep it sorted. The Binary Search Tree, or BST, is the classic attempt to have both: a linked, tree-shaped structure that keeps its elements in sorted order by the way it is built, so that searching, inserting and deleting all follow one short root-to-leaf path. This answer defines the BST precisely, explains how insertion and searching exploit its ordering property, describes the three depth-first traversals with particular attention to the in-order traversal the question asks for, and then gives a complete Python program that inserts the three given elements — 15, 10 and 20 — into an initially empty tree and displays them by in-order traversal, with the result verified by a hand trace.

Definition of a Binary Search Tree

A binary tree is a hierarchical structure of nodes in which each node holds a value and has at most two children, called the left child and the right child; the topmost node is the root, and a node with no children is a leaf. A Binary Search Tree is a binary tree that additionally satisfies the BST property at every node: all values in the left subtree of a node are smaller than the node's value, and all values in the right subtree are greater than the node's value, and both subtrees are themselves binary search trees. (Duplicate values are usually either forbidden or assigned consistently to one side by convention.) The recursive wording of the definition is not a formality: because every subtree is itself a BST, operations defined at the root work unchanged at every level, which is why the insertion, search and traversal code in this course is naturally recursive, in the style of Question 4. The practical reward of the property is that a search never needs to consider the whole tree. Standing at a node, a single comparison

disposes of half the remaining tree: if the target is smaller than the node, it cannot be anywhere in the right subtree, and if larger, it cannot be in the left subtree. The search therefore descends one level at a time, halving the candidates at each step exactly as binary search does, giving O(log n) operations in a balanced tree of n nodes — and degrading towards O(n) only in the worst case, where insertions in already sorted order make every node have a single right child and the tree degenerates into what is effectively a linked list. Keeping trees balanced against that danger is the business of advanced structures such as AVL and red-black trees; the plain BST defined here is their foundation.

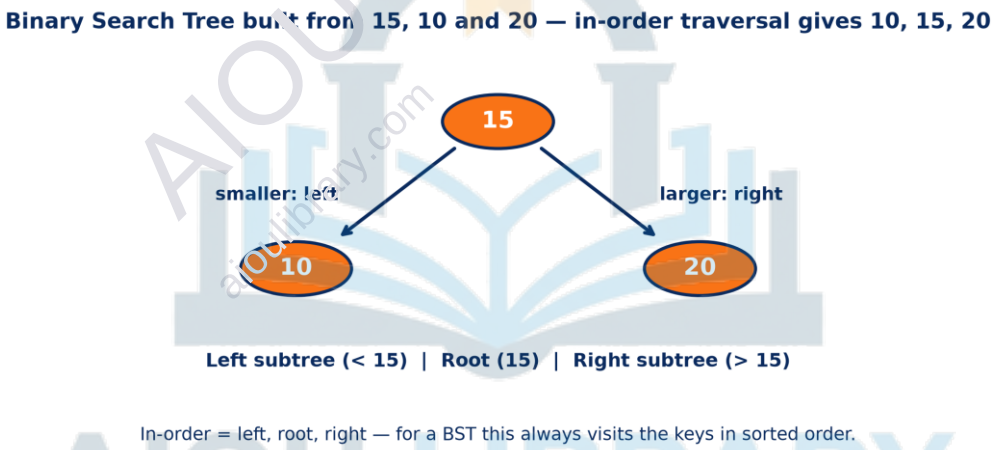

Figure 1: The BST built by inserting 15, then 10, then 20. The first value becomes the root; smaller values go left, larger values go right. In-order traversal visits the keys in sorted order: 10, 15, 20.

Insertion and Traversal in a BST

How Insertion Works

Insertion into a BST is simply an unsuccessful search that ends by attaching a leaf. Starting at the root, compare the new value with the current node: go left if it is smaller, right if it is larger.

Repeat at each node until a missing child — an empty place — is found, and attach the new node there. Because the comparisons at every step respect the BST property, the new node lands in the only position where the property remains true for the whole tree, and no existing node is ever moved. Insertion therefore costs O(log n) in a balanced tree, the cost of one root-to-leaf descent. The first value inserted into an empty tree has no comparisons to make and simply becomes the root — which is why, in the question's example, 15 is destined to be the root and the final shape of the tree is fixed by the insertion order: 10 must go somewhere in 15's left subtree and 20 somewhere in its right subtree. A different insertion order of the same values, such as 10, 15, 20, would produce a different, chain-shaped tree — an instructive reminder that a BST's shape, and hence its efficiency, depends on the order in which it was built.

Traversals, and Why In-Order Is Special

A traversal visits every node of a tree exactly once in a systematic order. The three standard depth-first traversals are distinguished by when the root of each subtree is visited relative to its children. Pre-order visits root, left subtree, right subtree, and is used for copying a tree and for prefix expression notation. Post-order visits left subtree, right subtree, root, and is used for deleting a tree safely and for evaluating expression trees, where children must be processed before their parent. In-order visits left subtree, root, right subtree. In-order traversal has a remarkable property on a BST: because every value in the left subtree is smaller than the root and every value in the right subtree is larger, visiting left, then root, then right necessarily produces the values in ascending sorted order. In-order traversal of a BST is therefore a sorting method in its own right — insert all values into a BST and read them back in-order and they

emerge fully sorted — and it is also the standard way of displaying a BST's contents, checking its correctness, and exporting its data in order. All three traversals visit every node once and so cost O(n) time; the in-order function below is recursive, each call handling a subtree by traversing its left part, printing its root and traversing its right part, with the empty tree (None) as the silent base case.

Table 1: The three depth-first traversals applied to the tree built in this question

Traversal	Order of visiting	Result on this tree (15, 10, 20)	Typical use
In-order	Left subtree, root, right subtree	10, 15, 20 (sorted)	Displaying a BST in order; sorting via a BST
Pre-order	Root, left subtree, right subtree	15, 10, 20	Copying a tree; prefix notation
Post-order	Left subtree, right subtree, root	10, 20, 15	Deleting a tree; evaluating expression trees

Program: Building the BST and Performing In-Order Traversal

The program below defines a Node class, a recursive insert function implementing the descend-and-attach algorithm, and a recursive in-order traversal function. It starts from an empty tree, inserts 15, 10 and 20 in the order given in the question, and then calls the traversal, which prints the elements on one line.

```
class Node:
```

```
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


def insert(root, value):
    if root is None:              # empty place found: attach here
        return Node(value)
    if value < root.data:         # smaller values go to the left
subtree
        root.left = insert(root.left, value)
    elif value > root.data:       # larger values go to the right
subtree
        root.right = insert(root.right, value)
    return root


def in_order(root):
    if root is not None:          # base case: empty tree, do nothing
        in_order(root.left)       # 1. traverse left subtree
        print(root.data, end=" ") # 2. visit (print) the root
        in_order(root.right)      # 3. traverse right subtree


# Build the BST from an empty tree
root = None
for value in [15, 10, 20]:
    root = insert(root, value)
```

```
print("In-Order Traversal of the BST:")
in_order(root)
```

Output of the program: In-Order Traversal of the BST: 10 15 20. The hand trace agrees. Inserting 15 into the empty tree makes it the root. Inserting 10 compares it with 15, finds it smaller, follows the left child, finds no node there, and attaches 10 as 15's left child. Inserting 20 compares it with 15, finds it larger, follows the right child, and attaches 20 as 15's right child. The resulting tree is exactly the one drawn in Figure 1, and it satisfies the BST property at its only non-leaf node: 10 < 15 < 20, with each subtree trivially a BST. The in-order traversal then calls itself on the left subtree, prints 10; returns and prints the root, 15; then calls itself on the right subtree and prints 20 -- producing the sorted sequence 10, 15, 20, as the theory promises. Two features of the code deserve a student's attention. First, insert returns the (possibly new) root of each subtree, and callers assign that return value back to the child pointer; this is what allows insertion into an empty tree and growth at the leaves to work through the same code path. Second, both functions are direct transcriptions of the recursive definition of a BST — which is why they are so short, and why the concept of recursion from Question 4 and the BST of this question belong in the same assignment: trees are recursive data, and recursive code is their natural language.

Conclusion

A Binary Search Tree is a binary tree in which, at every node, the left subtree holds smaller values and the right subtree holds larger values, so that searching and insertion each follow a

single root-to-leaf path, discarding half the tree at every comparison. Insertion attaches each new value as a leaf at the end of its own search path without disturbing any existing node, so the three given insertions build a tree rooted at 15, with 10 on the left and 20 on the right. In-order traversal — left subtree, root, right subtree — then displays those elements as 10, 15, 20, in sorted order, illustrating the general theorem that in-order traversal of any BST yields its keys sorted. Together, efficient ordered searching, insertion, and sorted display make the BST one of the central structures of this course, and the direct ancestor of the balanced trees and database indexes used in real systems.

AIOU LIBRARY
aioulibrary.com
AIOU LIBRARY