ALLAMA IQBAL OPEN UNIVERSITY



AIOU LIBRARY

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. 2

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.

AIOU LIBRARY aioulibrary.com

Question 1: Linear Search versus Binary Search; Binary Search for 70 (20 Marks)

Question as printed: Differentiate between Linear Search and Binary Search. Write a program to implement Binary Search on a sorted array [10, 23, 45, 70, 90, 100] to find the index of 70.

Introduction

Searching — finding whether a particular value is present in a collection, and if so where — is the most frequently performed operation in computing, and the difference between a good searching method and a poor one grows dramatically with the size of the data. Two elementary

methods frame the whole subject. Linear search makes almost no demands and almost no promises: it looks at the elements one after another until it finds the target or runs out of elements. Binary search makes one firm demand — the data must already be sorted — and in return makes a powerful promise: it eliminates half of the remaining elements with every single comparison. This answer differentiates the two methods point by point, explains the binary search algorithm and its conditions honestly, including the cost of keeping data sorted, and then gives a complete Python program that searches the given sorted array [10, 23, 45, 70, 90, 100] for the value 70, verified by a step-by-step trace showing the search space halving until the value is found at index 3.

Linear Search Described

Linear search, also called sequential search, starts at the first element and compares each element with the target in turn. If a comparison succeeds, it reports that element's position; if the end of the collection is reached without success, it reports that the target is absent. That is the entire algorithm, and its simplicity brings genuine advantages. It works on any collection whatever — sorted or unsorted, array or linked list — because it makes no assumption about order and needs no preparation of the data. It is trivial to program correctly and perfectly adequate, indeed preferable, for small collections, where the overhead of a cleverer method would exceed any saving. Its weakness is cost. In the worst case, when the target is the last element or absent altogether, every element must be examined, so the work grows in direct proportion to the size n of the collection: O(n) time, with an average of about n/2 comparisons for a successful search. Doubling the data doubles the search time, and at the scale of millions of records that proportionality is fatal. Linear search is also, notably, the only one of the two

methods that can be applied directly to a linked list, which supports no jumping to a middle element.

Binary Search Described

Binary search exploits sorted order to avoid looking at most of the data altogether. It maintains two boundaries, low and high, marking the portion of the array in which the target could still be present, and it begins with low at the first index and high at the last. At each step it examines the middle element, at index mid = (low + high) / 2. Three outcomes are possible. If the middle element equals the target, the search is over and mid is the answer. If the middle element is smaller than the target, then, because the array is sorted, every element at or before mid is also too small, so the whole left half is discarded at once by setting low = mid + 1. If the middle element is larger than the target, symmetrical reasoning discards the right half by setting high = mid - 1. The candidate region therefore at least halves on every iteration, and the search ends either when the target is found or when the region becomes empty (low passes high), proving the target absent. The number of comparisons is about log2 n — roughly 10 for a thousand elements, 20 for a million, 30 for a billion — giving O(log n) time, one of the most valuable growth rates in all of algorithm analysis. The conditions are equally definite: the data must be sorted, and it must support direct access to any position by index, which in practice means an array or similar structure rather than a linked list.

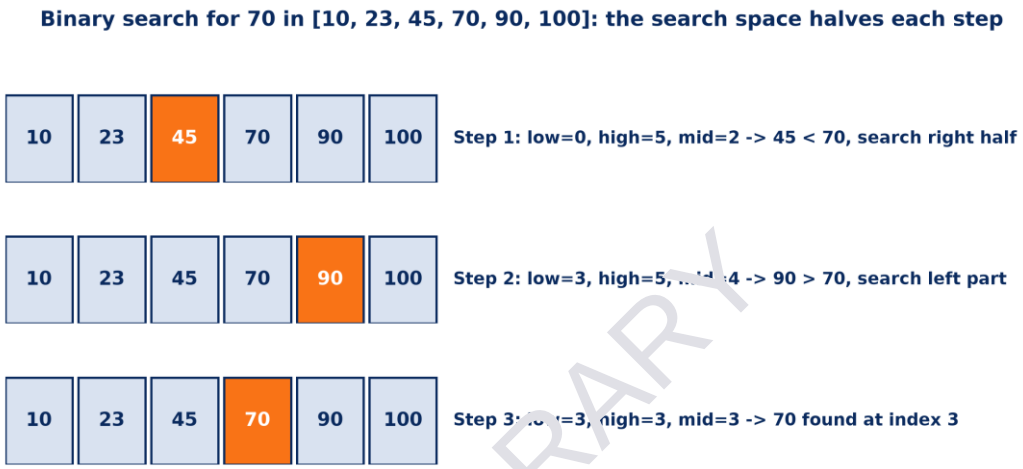

Figure 1: Binary search for 70 in [10, 23, 45, 70, 90, 100]. Each comparison discards half of the remaining candidates; 70 is found at index 3 in three steps.

Table 1: Linear search and binary search differentiated

Aspect	Linear Search	Binary Search
Basic method	Compares elements one by one from the start	Compares the middle element and discards half the data each step
Precondition	None; data may be in any order	Data must be sorted
Data structure	Works on arrays and linked lists alike	Needs random access by index — practically, an array
Worst-case time	O(n)	O(log n)
Average comparisons (successful)	About n/2	About log2 n
Best case	O(1) — target is the first element	O(1) — target is the middle element
Effect of doubling the data	Search takes about twice as	Search takes only one

	long	comparison more
Programming difficulty	Very simple	Slightly more care needed with low, high and mid
Suitable for	Small or unsorted collections; linked lists	Large, sorted collections searched repeatedly

Choosing Between Them: An Honest Comparison

Binary search is not simply the better method in every circumstance, and a good answer should state the trade-off plainly. Sorting is not free: arranging n elements costs O(n log n) comparisons, so if a collection will be searched only once, sorting it in order to binary-search it costs more than one linear search would have cost. Binary search wins when the data are already sorted for other reasons, or when they are searched many times, so the one-time sorting cost is spread over hundreds of fast searches. Linear search also remains the right tool while data are arriving unsorted, in small utility programs, and on linked structures. There is a close cousinship worth noticing between binary search and the Binary Search Tree of Assignment No. 1, Question 5: both discard half the candidates per comparison by exploiting order, the BST embodying in its shape the same decisions binary search makes with indices. Finally, correctness in binary search lives in small details that examiners like to test: mid must be recomputed inside the loop, low and high must move past mid (to mid + 1 and mid - 1), not merely to it, or the loop may never terminate, and the loop must continue while low <= high, so that a final one-element region is still examined — the very region in which the target of this question is found.

Program: Binary Search for 70, and the Trace

The program below follows the algorithm exactly. It keeps low, high and mid as described, prints information that makes the halving visible in teaching versions, and returns the index of the target or -1 if the target is absent. Indices in Python start at 0, so the elements of the given array occupy indices 0 to 5, and 70, the fourth element in ordinary counting, is expected at index 3.

```
def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2
        print(f"low={low}, high={high}, mid={mid},
arr[mid]={arr[mid]}")
        if arr[mid] == target:
            return mid              # target found
        elif arr[mid] < target:
            low = mid + 1           # discard the left half
        else:
            high = mid - 1          # discard the right half
    return -1                       # target not present

numbers = [10, 23, 45, 70, 90, 100]
target = 70
result = binary_search(numbers, target)
```

```
if result != -1:
    print("Element", target, "found at index", result)
else:
    print("Element", target, "not found in the array")
```

Output of the program: the trace lines show low=0, high=5, mid=2, arr[mid]=45; then low=3, high=5, mid=4, arr[mid]=90; then low=3, high=3, mid=3, arr[mid]=70; and the program reports 'Element 70 found at index 3'. Follow the reasoning at each step. First, mid = (0 + 5) // 2 = 2, and arr[2] is 45, which is less than 70, so — the array being sorted — 70 cannot be at index 2 or anywhere to its left, and low moves to 3, discarding three of the six candidates with one comparison. Next, mid = (3 + 5) // 2 = 4, and arr[4] is 90, which is greater than 70, so 70 cannot be at index 4 or to its right, and high moves to 3. The surviving region is the single index 3, and because the loop condition is low <= high, that region is still tested: mid = 3 and arr[3] is exactly 70, so the function returns 3. A linear search would have needed four comparisons to reach the same element (10, 23, 45, 70), and on this tiny array the difference is trivial; on a million sorted records the same comparison would be about 500,000 comparisons on average by linear search against at most 20 by binary search. Students should also test the program with an absent value such as 50: the region empties after examining 45 and 90's neighbourhood, low passes high, and the function returns -1, demonstrating that the same code proves absence as rigorously as it proves presence.

Conclusion

Linear search examines elements sequentially, needs no preparation and works anywhere, at O(n) cost; binary search requires sorted, indexable data and repays that requirement by halving the candidate region at every comparison, at O(log n) cost. Differentiating them is therefore not a matter of declaring a winner but of matching method to circumstances: unsorted or tiny data, or a linked list, call for linear search, while large sorted data searched repeatedly call decisively for binary search. The program and trace demonstrate the method on the exact data given, finding 70 at index 3 in three comparisons and discarding half the array at each of the first two — the halving behaviour from which all of binary search's power derives.

Question 2: Quick Sort — Working Mechanism and Program (20 Marks)

Question as printed: Explain the working mechanism of Quick Sort. Write a program to sort an array [29, 10, 14, 37, 13] using the Quick Sort algorithm and print the sorted array.

Introduction

Sorting — arranging a collection into a definite order, usually ascending — is a foundational operation: it prepares data for binary search, makes reports readable, groups equal values together and underlies everything from grade lists to database indexes. Among the many sorting algorithms, Quick Sort, devised by Tony Hoare in 1959, holds a special place: it is the algorithm most real systems actually use, because its average performance is excellent, it sorts largely in place, and its central idea is a beautiful example of the divide-and-conquer strategy.

That idea is easy to state and subtle to appreciate: choose one element as a pivot, rearrange the array so that everything smaller than the pivot stands before it and everything larger stands after it, and then note that the pivot is now in its final sorted position and the problem has split into two smaller, independent sorting problems of exactly the same kind. This answer explains the working mechanism — pivot selection, partitioning and recursion — analyses its performance honestly, including its worst case, and gives a complete Python program that sorts the given array [29, 10, 14, 37, 13], with a full trace of the computation.

The Working Mechanism of Quick Sort

Step 1: Choose a Pivot

Each stage of Quick Sort begins by selecting one element of the current sub-array to serve as the pivot, the standard against which all other elements in that sub-array will be compared. Common choices are the first element, the last element, the middle element, or a randomly chosen element; refined implementations take the median of the first, middle and last elements. The choice matters to performance, as the analysis below shows, because the pivot decides how evenly the problem splits. In the clear functional implementation used in the program below, the middle element of the list is taken as the pivot, a simple choice that behaves well on the data in this question and on already-sorted data, the traditional trap for first-element pivots.

Step 2: Partition the Array Around the Pivot

Partitioning is the heart of the algorithm and the only stage that moves data. The elements of the sub-array are divided by comparison with the pivot into groups: those smaller than the

pivot, those equal to it, and those greater than it. In the classic in-place version, two pointers sweep through the array exchanging misplaced elements, until a single scan leaves all smaller elements on the pivot's left and all larger elements on its right, with the pivot itself placed in the boundary position between them. The essential achievement of a partition is often under-appreciated by beginners: after it finishes, the pivot is in exactly the position it will occupy in the fully sorted array, and it never needs to move again, and furthermore no element will ever need to cross from one side to the other. One element is therefore finished, permanently, by each partition — which is real progress of a kind that, for example, a pass of bubble sort does not guarantee for its compared pairs. The cost of one partition is proportional to the size of the sub-array, O(n), since each element is compared with the pivot a constant number of times.

Step 3: Recurse on the Two Sub-Arrays

With the pivot fixed in its final place, Quick Sort calls itself on the left sub-array (the smaller elements) and on the right sub-array (the larger elements). Each of those calls chooses its own pivot, partitions around it, fixes one more element and splits again. The base case is a sub-array of size zero or one, which is already sorted and needs no work. As the recursion unwinds, no merging step is required at all — unlike merge sort, which must carefully combine its sorted halves — because partitioning has already guaranteed that everything on the left is smaller than everything on the right, so two internally sorted halves automatically form a sorted whole. This absence of a combine step is one of Quick Sort's quiet elegances, and it is possible only because its divide step does the hard work. The recursion depth, and hence the extra memory on the call stack, is about log n levels when partitions are balanced.

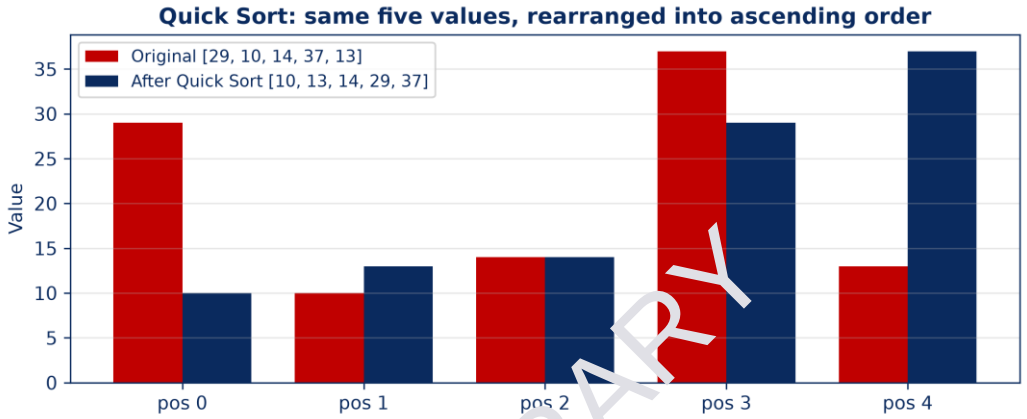

Figure 1: The given array before and after Quick Sort. Partitioning rearranges the same five values around successive pivots until the ascending order [10, 13, 14, 29, 37] is reached.

Performance: Average, Best and Worst Cases

If every pivot splits its sub-array roughly in half, the recursion forms about log2 n levels, each level performs O(n) total partitioning work across all its sub-arrays, and the total is O(n log n) — the same order as the best general comparison sorts, and the reason Quick Sort earns its name. This balanced behaviour is also the average behaviour for random data and reasonable pivot choices, and in practice Quick Sort is often faster than merge sort or heap sort at the same theoretical order, because its inner loop is simple, it works in place with only O(log n) stack memory, and its access pattern is friendly to processor caches. The worst case must be faced squarely: if the pivot is repeatedly the smallest or the largest element — as happens when the first element is chosen as pivot and the input is already sorted — each partition fixes one element and leaves a sub-array only one element smaller than before, the levels number n rather than log n, and the total degrades to O(n^2), no better than bubble sort. Real implementations defend against this with middle, random or median-of-three pivot selection, by switching to insertion sort for tiny sub-arrays, and in some library versions by monitoring

recursion depth and falling back to heap sort (the hybrid called introsort). A balanced judgement therefore describes Quick Sort as an O(n log n) average-case, in-place, unstable sort — unstable meaning that equal elements may change their relative order — whose worst case is a known and manageable risk rather than a secret.

Table 1: Quick Sort at a glance

Aspect	Detail
Strategy	Divide and conquer: partition around a pivot, then recurse on each side
Key operation	Partitioning, which places the pivot in its final sorted position
Best / average time	O(n log n) — balanced partitions
Worst-case time	O(n^2) — repeatedly extreme pivots, e.g. sorted input with a first-element pivot
Extra memory	O(log n) for the recursion stack (in-place versions); the teaching version below builds new lists
Stable?	No — equal elements may be reordered
Defences in practice	Random, middle or median-of-three pivot; insertion sort for small sub-arrays; introsort depth limit

Program: Sorting [29, 10, 14, 37, 13] with Quick Sort

The program below expresses the mechanism in its clearest functional form: a list of length one or less is returned unchanged (the base case); otherwise the middle element is chosen as pivot, the list is partitioned into three lists — smaller, equal and greater — and the result is Quick Sort of the smaller list, followed by the equal elements, followed by Quick Sort of the greater list. Production in-place versions partition within the original array to save memory, but the decisions, comparisons and final result are exactly the same, and this form makes every step visible to a student tracing by hand.

```
def quick_sort(arr):
    if len(arr) <= 1:                # Base case: already sorted
        return arr
    pivot = arr[len(arr) // 2]       # choose the middle element as
pivot
    smaller = [x for x in arr if x < pivot]
    equal   = [x for x in arr if x == pivot]
    greater = [x for x in arr if x > pivot]
    # Partition complete: sort each side recursively and join
    return quick_sort(smaller) + equal + quick_sort(greater)


numbers = [29, 10, 14, 37, 13]
print("Original array:", numbers)
sorted_numbers = quick_sort(numbers)
print("Sorted array:  ", sorted_numbers)
```

Output of the program: Original array: [29, 10, 14, 37, 13]; Sorted array: [10, 13, 14, 29, 37]. Now trace it. The list has five elements, so the first pivot is the middle element, arr[2] = 14. Partitioning compares every element with 14: the smaller group is [10, 13], the equal group is [14], and the greater group is [29, 37]. The pivot 14 is therefore fixed as the middle element of the final array — indeed, in the sorted result it stands at index 2, exactly where partitioning placed it conceptually. The left problem, [10, 13], is solved recursively: its pivot is 13, giving smaller [10], equal [13], greater [], which join to [10, 13]. The right problem, [29, 37], is solved the same way: its pivot is 37, giving smaller [29], equal [37], greater [], which join to [29, 37]. Joining the three finished pieces, [10, 13] + [14] + [29, 37], gives [10, 13, 14, 29, 37] — ascending throughout, containing exactly the original five values, nothing lost, nothing duplicated and nothing invented. Notice that only four non-trivial comparisons of structure were needed because each partition disposed of the whole list in one sweep; even on this tiny example the student can see the divide-and-conquer pattern that scales to millions of elements: fix the pivot, split the problem, let recursion finish the pieces, and spend no effort at all on merging.

Conclusion

Quick Sort works by repeatedly partitioning a sub-array around a chosen pivot, so that the pivot reaches its final position and all smaller elements stand to its left and all larger to its right, and then applying the same procedure recursively to each side until sub-arrays of size one make the whole array sorted. Its average and best performance of O(n log n), its in-place economy and its cache-friendly scanning explain its dominance in practice, while its O(n^2) worst case under repeatedly bad pivots explains the care real implementations take over pivot selection. Applied

to [29, 10, 14, 37, 13], the algorithm fixes 14 first, sorts [10, 13] and [29, 37] recursively, and prints the correctly sorted array [10, 13, 14, 29, 37] — the mechanism and the program agreeing exactly, as a hand trace confirms.

Question 3: Hash Tables and Collision Resolution by Linear Probing; Inserting Keys [12, 25, 35] (20 Marks)

Question as printed: What is a Hash Table, and how are collisions resolved using Linear Probing? Write a simple program to insert keys [12, 25, 35] into a hash table of size 10 using the hash function key % 10.

Introduction

Every searching method studied so far pays for finding a value with comparisons: linear search pays up to n of them, binary search and the binary search tree pay about log n. A hash table asks a more radical question — why search at all, if the position of a value can be calculated directly from the value itself? That is the hashing idea. A mathematical function, the hash function, converts each key into an index in an array, and the value is stored at that index; to find it later, the same function is applied again and the program goes straight to the calculated position, in O(1) average time regardless of how much data the table holds. The price of this near-magic is a new problem with a memorable name: since the number of possible keys is far larger than the number of positions, two different keys will sometimes be assigned the same index — a collision — and the table needs a disciplined way of coping. This answer defines the hash table and its components, explains collisions and the family of ways of resolving them,

treats linear probing in the detail the question requires, including its weakness, and gives a tested Python program inserting the given keys [12, 25, 35] into a table of size 10 with hash function key % 10, in which the third key deliberately produces the collision the question is designed to exhibit.

What a Hash Table Is

A hash table (also called a hash map or, in its commonest programming form, a dictionary) is a data structure that stores key-value pairs in an array, called the table or buckets array, whose size is chosen in advance. It has three working parts. First, the keys: unique identifiers such as a student's roll number, a customer's account number or a word in a dictionary, by which values will later be retrieved. Second, the hash function: a deterministic rule that takes a key and returns an index within the bounds of the table. Deterministic is essential — the same key must always produce the same index, or stored data could never be found again. The function given in this question, key % 10 for a table of size 10, is the simplest real example: the remainder on division by the table size automatically lies between 0 and 9, fitting the table exactly. For string keys, real hash functions first combine the characters into a large number and then reduce it modulo the table size. Third, the collision-resolution method, described in the next sections, which decides what happens when the calculated home position is already occupied. Insertion, search and deletion all follow the identical route: hash the key, go to the index, resolve any collision by the chosen method, and there perform the operation. Because no searching through unrelated data is involved, all three operations average O(1) time while the table is reasonably full — the property that makes hashing the fastest general-purpose lookup method

known, and the reason it underlies dictionaries in Python, database indexes, caches, symbol tables in compilers and countless other components.

Collisions: Why They Happen and How They Are Resolved

A collision occurs when two distinct keys hash to the same index. With key % 10, the keys 25 and 35 both produce 5, because both leave remainder 5 on division by 10; the keys 12 and 22 would collide in the same way at index 2. Collisions are not a sign of a defective hash function — with more possible keys than positions, the pigeonhole principle guarantees they must occur, and even an excellent function can only spread them out fairly, not abolish them. What distinguishes hash tables is how they resolve collisions, and two families of methods dominate. In separate chaining, each table position holds a small linked list (or similar structure) of all the pairs that hashed there, so colliding items coexist in the same bucket and a search examines that bucket's short chain; the table can even hold more items than it has positions. In open addressing, of which linear probing is the classic form, every item must occupy a table position of its own, so a colliding item is placed in a different, empty position found by a systematic search of the table itself. Quadratic probing and double hashing are refined members of the same family, differing only in how the alternative positions are generated. The question selects linear probing, so it is treated fully next; Table 1 places it among its alternatives.

Table 1: Collision-resolution methods compared

Method	Idea	Strength	Weakness
Separate chaining	Each position holds a list of all pairs hashing	Simple; table never 'fills up' in the same	Extra memory for lists and nodes; items are

	there	way; deletion is easy	not in the table array itself
Linear probing	On collision, try the next positions one by one: (h + 1), (h + 2), ... modulo size	Simple; excellent locality; all data in the array	Primary clustering; performance falls as the table fills
Quadratic probing	Try positions an increasing square away: h + 1, h + 4, h + 9, ...	Breaks up primary clustering	Can fail to find a free slot in some full-ish tables if carelessly designed
Double hashing	A second hash function decides the step size	Spreads items most evenly of the probing methods	More computation per step; implementation must ensure full coverage

Linear Probing Explained

In linear probing, insertion works as follows. Compute the home index h = hash(key). If that position is empty, place the item there and finish. If it is occupied, examine position (h + 1) % size; if that is occupied, examine (h + 2) % size, and so on, stepping forward one position at a time — wrapping from the end of the table back to the beginning through the modulo operation, exactly as in the circular queue of Assignment No. 1 — until an empty position is found, and place the item there. The sequence of positions examined, (h + i) % size for i = 0, 1, 2, ..., is called the probe sequence, and the number of positions examined is the probe count.

Searching must follow precisely the same sequence: hash the key, then compare the key stored at each probed position, stopping with success when the key is found and with failure when an empty position is reached — because if the key had ever been inserted, it would have been placed at or before that first empty position in its sequence. Deletion under probing needs equal care: simply emptying a position would break the search chains of later items that had probed past it, so deleted positions are marked with a special tombstone value that searches step over but insertions may reuse. Linear probing's characteristic weakness is primary clustering: once a run of occupied positions forms, any key hashing into the run is placed at its end, lengthening the run, so long runs grow ever more likely to attract further collisions, rather as traffic jams grow by absorbing the cars behind them. Keeping the load factor — the fraction of positions occupied — moderate, commonly below about 0.7, and rehashing into a larger table when it is exceeded, keeps probing short in practice; the little table in this question, ending with 3 of 10 positions filled, is comfortably within that discipline.

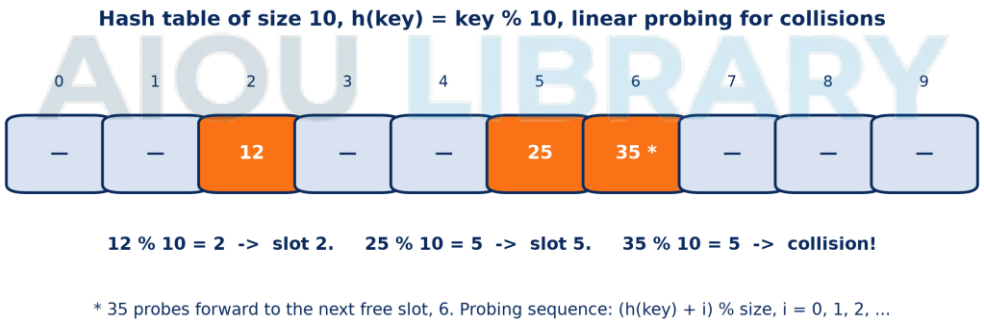

Figure 1: Keys 12, 25 and 35 in a table of size 10 with h(key) = key % 10. Key 35 collides with 25 at index 5 and is placed by linear probing in the next free position, index 6.

Program: Inserting [12, 25, 35] with Linear Probing

The program below builds a table of size 10 represented by a list of None values, defines insertion with linear probing exactly as described — printing each step so the collision and its resolution are visible — inserts the three given keys, and prints the final table. A short search function following the same probe sequence is included to show that the displaced key 35 can still be found where probing placed it.

```
SIZE = 10
table = [None] * SIZE


def hash_function(key):
    return key % SIZE


def insert(key):
    index = hash_function(key)
    original_index = index
    while table[index] is not None:      # collision: probe forward
        print(f"Collision for key {key} at index {index}, probing
next...")
        index = (index + 1) % SIZE        # linear probing step
        if index == original_index:
            print("Hash table is FULL")
            return
    table[index] = key
    print(f"Key {key} inserted at index {index}")
```

```
def search(key):
    index = hash_function(key)
    while table[index] is not None:
        if table[index] == key:
            return index
        index = (index + 1) % SIZE        # follow the same probe
sequence
    return -1

for key in [12, 25, 35]:
    insert(key)

print("Final hash table:", table)
print("Search 35 -> found at index", search(35))
```

Output and working: Key 12 is inserted at index 2, because 12 % 10 = 2 and position 2 is empty. Key 25 is inserted at index 5, because 25 % 10 = 5 and position 5 is empty. Key 35 first computes 35 % 10 = 5, finds position 5 occupied by 25 — the collision, announced in the output — and probes forward to position (5 + 1) % 10 = 6, which is empty, so 35 is inserted at index 6. The final table is therefore [None, None, 12, None, None, 25, 35, None, None, None], and the search for 35 retraces the same route — home index 5, mismatch, step to 6, found — returning index 6, which demonstrates the golden rule of probing: whatever path insertion took, search must be willing to take exactly the same path. Contrast this with the disordered alternative in which a colliding key were simply dropped or allowed to overwrite 25: hashing would then lose

data silently, the very failure collision resolution exists to prevent. Note also what the example teaches about the hash function itself: remainders group keys by their last digit, so any set of keys ending in the same digit collides at one home index; production hash functions work much harder to scramble such patterns, and production tables are usually given a prime size rather than the round 10 chosen here for teaching clarity.

Conclusion

A hash table stores key-value pairs at array positions calculated from the keys by a deterministic hash function, buying O(1) average insertion, search and deletion at the price of inevitable collisions when distinct keys share a home index. Linear probing resolves a collision by stepping forward one position at a time, wrapping around the table, until an empty position accepts the displaced item, and search faithfully retraces the same probe sequence. In the required example, 12 takes index 2, 25 takes index 5, and 35 — colliding with 25 at index 5 — is placed by probing at index 6, and can be found there again by the same route. Simple, local and effective at moderate load factors, linear probing remains the standard first method for teaching collision resolution and a component of many real hash tables, tempered in practice by attention to clustering and timely rehashing.

Question 4: Graphs, Adjacency Lists and Breadth-First Search (BFS) (20 Marks)

Question as printed: What is a Graph in data structures? Write a program to represent a graph using an Adjacency List for 4 vertices (0 to 3) with edges (0-1, 0-2, 1-3, 2-3). Also, write a function to perform Breadth-First Search (BFS) starting from vertex 0.

Introduction

Arrays, stacks, queues and trees all impose a strict shape on data: a line, a pile, a waiting line, a hierarchy. Much of the world's data refuses those shapes. Cities are joined by roads in a web, computers by network links, people by friendships that cross and recross, and web pages by hyperlinks pointing in every direction. The data structure designed for such many-to-many relationships is the graph, the most general structure in this course — so general that a tree can be defined as a special kind of graph, and a linked list as an even more special one. This answer defines the graph and its vocabulary, compares the two standard ways of representing a graph in memory and explains why the question's choice, the adjacency list, is usually preferred, and then gives a complete Python program that builds the specified graph of four vertices with four given edges as an adjacency list and performs a breadth-first search starting from vertex 0, with the traversal traced level by level.

What a Graph Is

A graph G is a pair (V, E), where V is a set of vertices (also called nodes) and E is a set of edges, each edge joining a pair of vertices. In the graph of this question, V = {0, 1, 2, 3} and E = {(0, 1),

(0, 2), (1, 3), (2, 3)}. Several distinctions organise the subject. In an undirected graph an edge is a two-way connection with no orientation, as a road between two towns; in a directed graph (digraph) each edge has a direction, drawn as an arrow, as a hyperlink points from one page to another. In a weighted graph each edge carries a number, its weight — a distance, a cost, a journey time — which algorithms such as shortest-path methods take into account; in an unweighted graph all edges count alike, as in this question. Two vertices joined by an edge are called adjacent or neighbours, and the degree of a vertex is the number of edges touching it. A path is a sequence of vertices connected by edges, and a graph is connected if a path exists between every pair of vertices — the given graph is connected, since vertex 0 reaches 3 by two different routes, 0-1-3 and 0-2-3, which also shows it contains a cycle, the closed loop 0-1-3-2-0. Graphs model road and rail networks, computer networks and the Internet, social networks, maps for route-planning, dependencies between tasks in a project, states and transitions in software, and the web itself; the algorithms run on them — traversal, shortest path, minimum spanning tree — are among the most practically valuable in all of computing.

Representing a Graph: Adjacency Matrix and Adjacency List

To store a graph, a program must record, for the vertices, which other vertices each one is connected to. The adjacency matrix does this with a two-dimensional array: rows and columns are both indexed by vertices, and the entry at row i, column j is 1 (or the edge's weight) if an edge joins i and j, and 0 otherwise. The matrix is simple and answers 'is there an edge between i and j?' in O(1), but it costs V-squared memory always — a million-vertex network, most of whose vertices have only a handful of connections, would demand a trillion entries, almost all of them zero. The adjacency list, which the question asks for, instead keeps, for each vertex, a

list of just its neighbours: in this question, 0: [1, 2], 1: [0, 3], 2: [0, 3], 3: [1, 2]. Its memory cost is O(V + E), proportional to what actually exists, which for the sparse graphs of the real world — road networks, social graphs, the web — is dramatically smaller. Checking one specific edge costs O(degree) in the worst case, a small price most applications gladly pay, and listing all neighbours of a vertex, which traversal algorithms do constantly, is as fast as it can possibly be, since the neighbours are stored together ready-made. One implementation detail matters: the graph in this question is undirected, so every edge is stored twice, once in each endpoint's list — edge 0-1 puts 1 in 0's list and 0 in 1's list — because adjacency truly runs both ways; a directed graph would store each edge once only, in its starting vertex's list.

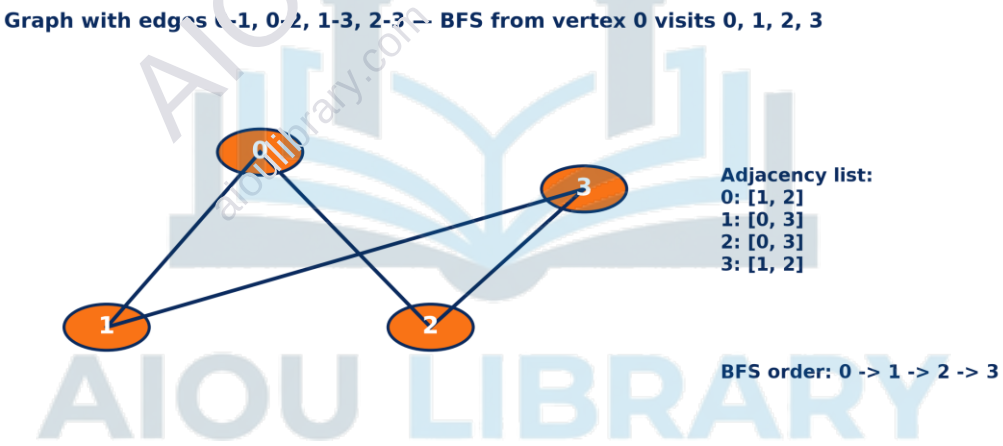

Figure 1: The graph of this question and its adjacency list. Breadth-first search from vertex 0 visits the vertices level by level in the order 0, 1, 2, 3.

Table 1: Adjacency matrix and adjacency list compared

Aspect	Adjacency Matrix	Adjacency List
Storage	O(V^2), always	O(V + E) — proportional to actual edges

Edge lookup (is i joined to j?)	O(1)	O(degree of i)
Listing all neighbours of a vertex	O(V) — the whole row must be scanned	O(degree) — exactly the stored neighbours
Best suited to	Dense graphs; very frequent edge lookups	Sparse graphs — most real-world networks; traversals
Adding a vertex	Expensive: a new row and column	Easy: a new empty list

Breadth-First Search (BFS)

A traversal visits the vertices of a graph systematically from a starting vertex, and breadth-first search does so level by level: it visits the start vertex, then all of the start vertex's neighbours, then all of their unvisited neighbours, expanding outwards like ripples on a pond, so vertices are visited in order of their distance (in edges) from the start. BFS is built on a queue — the FIFO structure of Assignment No. 1, Question 3 — and its discipline has three rules. Mark the start vertex visited and enqueue it. While the queue is not empty, dequeue the front vertex, visit (output) it, and enqueue every neighbour of it that has not yet been marked, marking each neighbour at the moment it is enqueued, not when it is later dequeued. Marking at enqueue time is the detail beginners most often miss, and it matters: a vertex reachable from two others — vertex 3 in this question, neighbour to both 1 and 2 — would otherwise be enqueued twice and visited twice. The visited set guarantees each vertex enters the queue exactly once, so BFS costs O(V + E) time: every vertex is queued once and every adjacency-list entry is examined once. Beyond mere visiting, BFS has a prized property on unweighted graphs: the first time it

reaches a vertex, it has arrived by a path with the fewest possible edges, so BFS computes shortest paths in unweighted graphs, powers level-by-level features such as finding all friends within two connections, and drives web crawlers and network broadcasting. Its sibling, depth-first search, uses a stack (or recursion) instead of a queue and dives down one path as far as possible before backtracking; the queue-versus-stack contrast between the two traversals is a favourite examination point, and a tidy summary of how much behaviour follows from the choice of one elementary structure over another.

Program: Adjacency List Representation and BFS from Vertex 0

The program below uses a Python dictionary mapping each vertex to a list of neighbours — the direct transcription of an adjacency list — adds each given edge in both directions because the graph is undirected, prints the representation, and then runs a BFS implemented with a queue (collections.deque, whose popleft operation is efficient) and a visited set, exactly as specified above.

```
from collections import deque


# Create an adjacency list for 4 vertices (0 to 3)
graph = {0: [], 1: [], 2: [], 3: []}


def add_edge(u, v):
    graph[u].append(v)
    graph[v].append(u)    # undirected edge: store in both lists


# Add the given edges
```

```
for u, v in [(0, 1), (0, 2), (1, 3), (2, 3)]:
    add_edge(u, v)


print("Adjacency List Representation:")
for vertex in graph:
    print(vertex, "->", graph[vertex])


def bfs(start):
    visited = set([start])          # mark the start vertex visited
    queue = deque([start])          # and enqueue it
    order = []
    while queue:
        vertex = queue.popleft()          # dequeue the front vertex
        order.append(vertex)              # visit it
        for neighbour in graph[vertex]:
            if neighbour not in visited:
                visited.add(neighbour)    # mark when enqueueing
                queue.append(neighbour)   # enqueue the neighbour
    return order


print("Breadth-First Search starting from vertex 0:")
print(bfs(0))
```

Output of the program: the adjacency list prints as 0 -> [1, 2], 1 -> [0, 3], 2 -> [0, 3], 3 -> [1, 2], and the BFS prints [0, 1, 2, 3]. Trace the search to see the level discipline at work. Initially vertex 0 is marked and queued: queue [0]. Dequeue 0 and visit it; its neighbours 1 and 2 are unvisited,

so both are marked and enqueued: queue [1, 2]. Dequeue 1 and visit it; its neighbours are 0 (already visited) and 3 (unvisited), so 3 is marked and enqueued: queue [2, 3]. Dequeue 2 and visit it; its neighbours are 0 and 3, both already marked — and here the mark-at-enqueue rule earns its keep, because 3 is still waiting in the queue, not yet visited, yet is correctly not enqueued a second time. Dequeue 3 and visit it; both its neighbours are marked, so nothing is added, the queue empties, and the search ends. The visit order 0, 1, 2, 3 displays BFS's signature plainly: distance 0 first (vertex 0 itself), then both vertices at distance 1 (1 and 2, in list order), then the single vertex at distance 2 (vertex 3) — and the path by which 3 was discovered, through vertex 1, has two edges, which no shorter path can beat, since 3 is not a direct neighbour of 0. Had the search started at vertex 3 instead, the equally correct BFS order would have been 3, 1, 2, 0: traversal orders describe a journey from a starting point, not a property of the graph alone.

Conclusion

A graph is a set of vertices joined by edges, directed or undirected, weighted or unweighted, and it is the natural model for networks of every kind, from roads and computers to friendships and web pages. The adjacency list represents it economically — O(V + E) memory — by storing for each vertex exactly its neighbours, which for the given graph yields 0: [1, 2], 1: [0, 3], 2: [0, 3] and 3: [1, 2], each undirected edge recorded both ways. Breadth-first search then explores that representation level by level using a queue and a visited set, visiting each vertex and edge once in O(V + E) time, and from vertex 0 produces the order 0, 1, 2, 3 — discovering every vertex by a shortest possible route in edges as it goes. Representation and algorithm fit together exactly: the adjacency list exists to hand a traversal its neighbours efficiently, and BFS

is the traversal that turns those neighbour lists into distances, orders and, in larger applications, routes.

Question 5: Tree Structures for Hierarchical Data; Building a BST from an Array and Searching for a Value (20 Marks)

Question as printed: How are Tree structures used in organizing hierarchical data? Write a program that takes an array of numbers and builds a Binary Search Tree from it. Then, write a function to search for a specific value (e.g., 40) in that tree and return whether it is found or not.

Introduction

Some data are flat — a list of marks, a queue of jobs — but a great deal of the world's most important data are inherently nested: a company is divided into divisions, which contain departments, which contain teams; a book is divided into parts, chapters and sections; a computer's storage is divided into drives, folders within folders, and files. Flat structures force such data into awkward workarounds, duplicating parent information in every row or trusting fragile naming conventions. The tree, the hierarchical data structure introduced through the Binary Search Tree in Assignment No. 1, represents nesting directly: one distinguished root, each item holding its own children beneath it, and containment expressed by position rather than by repetition. This answer explains how trees organise hierarchical data, drawing on the examples a student meets daily, revisits the vocabulary and the balance question that govern every tree's efficiency, and then gives a complete Python program that builds a Binary Search

Tree from an array of numbers and searches it for the specified value 40, returning whether it is found — with the construction and both a successful and an unsuccessful search traced by hand.

How Trees Organise Hierarchical Data

A tree organises data by parent-child relationship. The single item at the top, with no parent, is the root; every other item has exactly one parent and may have any number of children (at most two in a binary tree); items with no children are leaves; and the items under one parent form that parent's subtree, which is itself a tree in miniature. Position in this structure carries meaning that a flat list cannot express. Ancestry answers 'what does this belong to?' by walking upwards — a file's full path, a subordinate's chain of command, a section's book. Descendants answer 'what does this contain?' by walking downwards — everything in a folder, everyone under a manager, every species in a genus. The depth of an item (its distance from the root) measures its level in the hierarchy, and the height of the tree (the longest root-to-leaf path) governs how many steps any top-down operation can take, which is why balanced trees, whose height stays near log n, are so prized. Crucially, information shared by a whole group is stored once, at the group's node: a department's name and budget live in the department node, not copied into every employee record beneath it, so an update is made in exactly one place and inherited by position — the same war on uncontrolled redundancy that the database part of this syllabus wages with keys and normalisation, fought here by structure instead of by schema.

Everyday and Computing Examples

Once the pattern is recognised, it is visible everywhere, and an examination answer gains strength from naming precise instances. A file system is a tree: the root directory contains folders, folders contain subfolders and files, and a path such as Documents/Assignments/DataStructures is literally a record of a descent from root to leaf; deleting a folder deletes its subtree, which is why post-order traversal (children before parent) is the deletion order. An organisation chart is a tree, the chief executive at the root, managers as internal nodes and individual workers largely at the leaves, used for reporting lines and for rolling up figures such as salaries or headcounts from leaves to root. A family tree and the biological classification of living things (kingdom, phylum, class, order, family, genus, species) are trees of descent and of containment respectively. In computing itself the examples multiply: the Document Object Model of a web page is a tree of elements nested inside elements, and the HTML a student writes is a tree written out flat; a compiler holds each expression as a tree, operators as internal nodes and numbers as leaves, evaluating it by post-order traversal; a decision tree in machine learning routes each case from a root question down a path of tests to a leaf prediction; a university's own structure — university, faculties, departments, programmes, courses — is the hierarchy this very assignment travels through on its way to a tutor. In each case the tree succeeds for the same three reasons: containment is expressed once, by position; groups can be processed as units by processing a subtree; and finding an item is a single guided descent rather than a scan of everything.

Table 1: Hierarchical data held in trees — examples

Domain	Root	Internal nodes	Leaves	What the tree expresses
File system	Root directory / drive	Folders and subfolders	Files	Containment; a path is a root-to-leaf descent
Organisation	Chief executive	Divisions, departments, managers	Individual employees	Reporting lines; totals roll up from leaves
Book / document	The book or document	Parts and chapters	Paragraphs / sections at the lowest level	Structure; a table of contents is a tree listing
Web page (DOM)	The html element	Structural elements (body, div, table)	Text and image content	Nesting of the page's components
Expression in a compiler	The main operator	Sub-operators	Numbers and variables	Order of evaluation; post-order computes the value

From General Trees to the Binary Search Tree

A general tree allows any number of children per node and organises by containment; a binary tree restricts each node to at most two children; and a Binary Search Tree, defined in Assignment No. 1, Question 5, adds the ordering rule that a node's left subtree holds smaller

values and its right subtree larger values. That rule converts a hierarchy into a searching machine: to find a value, start at the root, go left if the target is smaller, right if it is larger, and repeat until the value is found or an empty place proves it absent — one decision per level, about log n decisions in a balanced tree, and never a look at irrelevant subtrees. Building a BST from an array simply repeats BST insertion for each array element in order: the first element becomes the root, and each subsequent element descends to the empty position its comparisons select. Two warnings, carried over from Assignment No. 1, belong in a complete answer. First, the shape — and therefore the efficiency — depends on insertion order: the well-spread array used in the program below builds a handsome balanced tree, while an already sorted array would build a degenerate chain of right children, effectively a linked list with O(n) search, which is why self-balancing trees (AVL, red-black) exist in production systems. Second, the recursive formulation of insertion and search is not an affectation: subtrees are trees, so functions that call themselves on subtrees transcribe the definition directly, and their base case — meeting None where a child would be — is precisely how absence is detected.

Program: Building a BST from an Array and Searching for 40

The program below builds a BST from the array [50, 30, 70, 20, 40, 60, 80] — chosen so that 40, the value the question names as the example target, is genuinely present and interestingly placed, neither at the root nor at a leaf's most obvious position — displays the tree's in-order traversal as a correctness check (it must print the array's values in sorted order), and then searches for 40 and for 45, returning and reporting Found or Not Found for each.

```
class Node:
    def __init__(self, data):
```

```
        self.data = data
        self.left = None
        self.right = None


def insert(root, value):
    if root is None:
        return Node(value)
    if value < root.data:
        root.left = insert(root.left, value)
    elif value > root.data:
        root.right = insert(root.right, value)
    return root


def build_bst(array):
    root = None
    for value in array:
        root = insert(root, value)
    return root


def search(root, target):
    if root is None:
        return False                    # empty place reached: not present
    if target == root.data:
        return True                     # found at this node
    elif target < root.data:
        return search(root.left, target)    # look in the left subtree
    else:
```

```
        return search(root.right, target)  # look in the right subtree


def in_order(root):
    if root is not None:
        in_order(root.left)
        print(root.data, end=" ")
        in_order(root.right)


numbers = [50, 30, 70, 20, 40, 60, 80]
bst_root = build_bst(numbers)

print("In-Order traversal (sorted check):")
in_order(bst_root)
print()

for value in [40, 45]:
    if search(bst_root, value):
        print(value, "-> Found in the BST")
    else:
        print(value, "-> Not Found in the BST")
```

Output of the program: the in-order traversal prints 20 30 40 50 60 70 80 — the array's values in perfect ascending order, proving the tree satisfies the BST property throughout — and the searches report '40 -> Found in the BST' and '45 -> Not Found in the BST'. Trace the construction briefly: 50 becomes the root; 30 goes to its left; 70 to its right; 20 descends past 30

to become 30's left child; 40 descends past 30 (40 is larger) to become 30's right child; 60 and 80 become 70's left and right children, completing a perfectly balanced tree of height 3. Now trace the required search for 40. At the root, 50: 40 is smaller, so the entire right subtree (70, 60, 80) is dismissed without a glance, and the search descends left to 30. At 30: 40 is larger, so the left subtree (just 20) is dismissed, and the search descends right — to 40 itself, and returns True. Two comparisons examined two nodes out of seven and conclusively discarded four others; that economy is the BST property doing its work. The search for 45 follows the identical path — left at 50, right at 30, right again at 40 (since 45 is larger) — and arrives at the empty right child of 40, where the base case returns False: absence is proved, not merely not noticed, because had 45 been inserted anywhere in this tree, the insertion rules would have placed it in exactly that empty position. Iterative versions of the same search, using a loop and a moving current pointer instead of recursion, behave identically; the recursive form is shown because it mirrors the definition and matches the style of this course.

Conclusion

Tree structures organise hierarchical data by expressing containment and belonging through parent-child position: one root, exactly one parent per item, subtrees processible as units, shared group information stored once at the group node, and any item reachable by a single descent from the root — the pattern of file systems, organisation charts, document structures and classifications alike. The Binary Search Tree specialises the tree with an ordering rule that turns descent into search. The program demonstrates the complete workflow the question requires: an array of numbers is folded, element by element, into a balanced BST (verified by its sorted in-order traversal), and a search function then follows the ordering rule down a single

path to report 40 Found — and, with equal rigour, an absent value Not Found. Hierarchy for organisation, order for retrieval: that combination is why trees stand at the centre of data structures, and why this course ends its second assignment with them.

AIOU LIBRARY
aioulibrary.com
AIOU LIBRARY