INDIRAGANDHINATIONALOPENUNIVERSITY

Regional Centre Delhi-3

Format for Assignment Submission

For Term End Exam June/December- JUNE **(Year)** 2025

(Please read the instructions given below carefully before submitting assignments)

1. **Name of the Student** : Ashu Chaurasiya
2. **Enrollment Number** : 2501321326
3. **Programme Code** : MCA_NEW
4. **Course Code** : ~~LSC-38046~~ MCS-208
   (Use this format course-wise separately)
5. **Study Centre Code** : LSC-38046
6. **Name of the Study Centre With complete address** : RAJDHANI COLLEGE RAJA GARDEN NEW DELHI
7. **Mobile Number** : 8448713694
8. **E-mail ID** : Intermezzobrilliance@gmail.com
9. **Details if this same assignment has been submitted anywhere else also** : No
10. **Above information is cross checked and it is correct: Yes/No** ✓

Date of Submission: 01-06-25

(Signature of the student)

A. Important Instructions:-

1. Please do not send any assignment at any email of the Regional Centre, it will not be considered.
2. Please avoid duplicacy. Do not re-submit the same assignment anywhere else or by any other means.
3. About the mode of submission of assignments, pl wait for instructions from IGNOU Hqtrs. As soon as we shall come to know, we will share it with all.
4. Please do not use plastic covers. Use plain A4 size pages for assignments for uniformity and better management with this cover page format on each assignment.
5. Please write your name and enrollment no. at the bottom of each page of your assignment.
6. Please retain a photocopy set of assignment submitted with you for record(may be asked to submit at later stage) and also keep the assignment submission receipt in safe custody.
7. If assignment awards are not updated in your Grade Card **within next 09 months**, please write to us at **rcdelhi3@ignou.ac.in giving your complete details and attaching the proof of assignment submission.**
8. Assignment Question Paper can be downloaded from: https://webservices.ignou.ac.in/assignments/

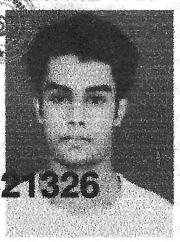
B. Compulsory sequence of the Assignment Set:

1. Duly Filled in Assignment Submission Cover Page (This Format Page).
2. Copy of IGNOU Identity Card.
3. Print out of valid/applicable assignment question paper.
4. Handwritten Assignment, written on both the sides of page (preferably plain A4 size).

ignou
THE PEOPLE'S
UNIVERSITY

इंदिरा गांधी राष्ट्रीय मुक्त विश्वविद्यालय
मैदान गढ़ी, नई दिल्ली - 110068
Indira Gandhi National Open University
Maidan Garhi,New Delhi - 110068

IGNOU - Student Identity Card



2501321326

Enrolment Number :	2501321326
RC Code :	38: DELHI 3 Naraina
Name of the Programme :	MCA_NEW : Master of Computer Applications
Name :	ASHU CHAURASIYA
Father's Name :	VIJAY
Address :	Flat No. 270,G, F Block-14, Pocket 13, Sector-20 RNORTH WEST
Pin Code :	110086

Instructions :
1. This card should be produced on demand at the Study Center, Examination Center or any other Establishment of IGNOU to use its facilities.
2. The facilities would be available only relating to the Programme/course for which the student is registered.
3. This ID Card is generated online. Students are advised to take a color print of this ID Card and get it laminated.
4. The student details can be cross checked with the QR Code at www.ignou.ac.in

Registrar
Student Registration Division

Course Code	:	**MCS-208**
Course Title	:	**Data Structures and Algorithms**
Assignment Number	:	**PGDCA_NEW(II)/208/Assign/2025**
Maximum Marks	:	**100**
Weightage	:	**25%**
Last Dates for Submission	:	**30th April 2025 (for January Session)**

There are four questions in this assignment, which carry 80 marks. Each question carries 20 marks. Rest 20 marks are for viva voce. All algorithms should be written nearer to C programming language. You may use illustrations and diagrams to enhance the explanation, if necessary. Please go through the guidelines regarding assignments given in the Programme Guide for the format of presentation.

Q1: What is a Doubly Linked Circular List? What are its advantages and disadvantages? Give a scenario where its application is appropriate. Justify your answer. **(20 Marks)**

Q2: What is a Tree? How does it differ from a Binary Tree? Is it possible to convert a Tree to a Binary Tree? If yes, then, explain the process with an example. **(20 Marks)**

Q3: What are Red Black Trees? How do they differ from Splay Trees? What are their applications? **(20 Marks)**

Q4: Write a short note on the recent developments in the area of finding shortest path between two nodes of a Graph. Make necessary assumptions. **(20 Marks)**

Enrollment No. 2501321326

Question no. 1: What is a Doubly Linked Circular List? What are its advantages and disadvantages? Give a scenario where its application is appropriate. Justify your answer.

Answer:

Doubly Linked Circular List: A Doubly Linked Circular List is a type of linked list where each node has three components: a data field, a pointer (next) that references the subsequent node in the sequence, and a pointer (prev) that references the preceding node. The "circular" aspect means that the next pointer of the last node points back to the first node, and conversely, the prev pointer of the first node points to the last node. This structure eliminates the concept of a NULL pointer at the ends of the list, as it forms a continuous loop.

Advantages: One significant advantage of a doubly linked circular list is its bi-directional traversal capability. Unlike singly linked lists, you can traverse the list in both forward and backward directions, which is highly efficient for operations that require inspecting both preceding and succeeding elements. Another key benefit is the efficiency of insertion and deletion operations. Once the specific node or its adjacent nodes are located, inserting or deleting a node can be done in constant time (O()) by simply updating a few pointers. Furthermore, the absence of NULL pointers at the ends of the list can simplify certain algorithms, as there's no need for special NULL checks when traversing, making it ideal for applications that require continuous looping. Lastly, accessing the last

immediate from the head (by following head->prev), offering O(1) access time.

Disadvantages: The primary disadvantage is increased memory overhead. Each node requires two pointer fields (next and prev) instead of just one, meaning each node consumes more memory compared to its singly linked list counterpart. This can be a concern for applications dealing with a very large number of nodes or constrained memory environments. Additionally, implementing operations becomes slightly more complex. While efficient, insertion and deletion involve updating four pointers (two for the new/deleted node, and two for its neighbors), which is more intricate than in singly linked lists and can be more prone to off-by-one or null pointer errors if not handled carefully. For very small lists, the overhead of managing these extra pointers might negate any benefits.

Scenario for Application: A highly appropriate scenario for the application of a Doubly Linked Circular List is in the implementation of a round-robin task scheduler in an operating system or an embedded system.

Justification: In a round-robin task scheduler, multiple tasks are given slices of CPU time in a sequential, repeating fashion. When a task finishes its time slice or explicitly yields the CPU, the scheduler needs to quickly determine the next task to run. A doubly linked circular list fits this need perfectly. The circular nature of the list naturally models the round-robin behavior, allowing the scheduler to continuously loop through the tasks. The bi-directional traversal is beneficial because the

scheduler might need to easily jump to the next task in sequence (forward) or, in certain scenarios (e.g., if a task needs to be re-queued immediately or if task priority changes), easily reference the previously executed task (backward). Moreover, the efficient insertion and deletion capabilities are crucial for dynamic task management: new tasks can be added to the queue when they are created, and completed tasks can be efficiently removed, all without significantly impacting the performance of the scheduler. This structure ensures a smooth, continuous, and responsive management of concurrent tasks within the system.

Question no. 2: What is a Tree? How does it differ from a Binary Tree? Is it possible to convert a Tree to a Binary Tree? If yes, then, explain the process with an example.

Answer:

What is a Tree? In computer science, a Tree is a non-linear, hierarchical data structure composed of nodes connected by edges. It begins with a unique node called the "root," which has no parent. Every other node in the tree has exactly one parent node. Nodes can have zero or more child nodes. Trees are fundamental for representing hierarchical relationships, such as organizational structures, file systems, or the syntax of programming languages in compilers (parse trees).

How does it differ from a Binary Tree? The fundamental difference between a general tree and a binary tree lies in the constraint on the number of children a node can possess. In a general tree, a node has the

flexibility to have any number of children — it could have zero (making ita leaf node), one, two, or even many more. Conversely, a binary treeimposes a strict limitation: each node can have at most two children These children are conventionally designated as the "left child" and the "right child." This structural constraint is what gives binary trees their specific properties and allows for specialized algorithms.

Is it possible to convert a Tree to a Binary Tree? If yes, then, explain the process with an example. Yes, it is indeed possible to convert any general tree into an equivalent binary tree. This transformation is a common technique known as the "Left-Child Right-Sibling" (LCRS) representation, or sometimes referred to as the "Binary Tree Representation of General Trees." This conversion is highly useful because it allows algorithms and data structures designed for binary trees to be applied to problems involving general trees.

Process (Left-Child Right-Sibling Representation): The conversion process involves re-interpreting the pointers of each node in the original general tree to fit the binary tree structure:

Left Child Pointer: In the new binary tree, the left child pointer of any given node will point to that node's first child from the original general tree.

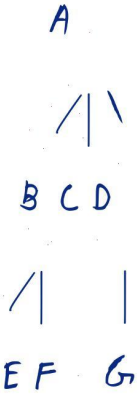
Right Child Pointer: The right child pointer of a node in the binary tree will point to that node's next sibling from the original general tree.

Root: The root node of the original general tree remains the root node of the newly formed binary tree.

Conceptually, this means that in the transformed binary tree, all children of an original general tree node are linked horizontally using the "right-child" pointers, starting from the leftmost child (which is pointed to by the "left-child" pointer of the parent).

Example:

Let's consider the following general tree:



Here, 'A' is the root. 'B', 'C', 'D' are children of 'A'. 'E', 'F' are children of 'B'. 'G' is a child of 'D'. 'C', 'E', 'F', 'G' are leaf nodes.

Now, let's convert this to its Left-Child Right-Sibling (LCRS) binary tree representation:

Node A:

Its first child in the original tree is 'B'. So, in the binary tree, A's left pointer will point to 'B'.

'A' has no siblings. So, in the binary tree, A's right pointer will be NULL.

Node B:

Its first child in the original tree is E'. So, B's left pointer will point to E'.

Its next sibling in the original tree is C'. So, B's right pointer will point to C'.

Node C:

C' has no children in the original tree. So, C's left pointer will be NULL.

Its next sibling in the original tree is D'. So, C's right pointer will point to D'.

Node D:

Its first child in the original tree is G'. So, D's left pointer will point to G'.

D' has no siblings. So, D's right pointer will be NULL.

Node E:

E' has no children. So, E's left pointer will be NULL.

Its next sibling in the original tree is F'. So, E's right pointer will point to F'.

Node F:

F' has no children. So, F's left pointer will be NULL.

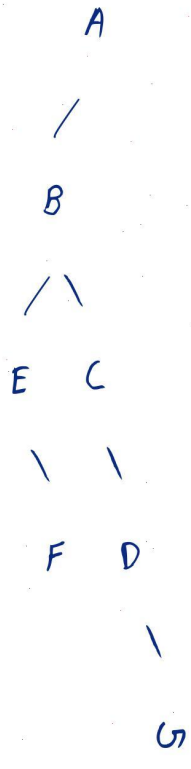
F' has no siblings. So, F's right pointer will be NULL.

Node G:

G' has no children. So, G's left pointer will be NULL.

G has no siblings. So, G's right pointer will be NULL.

The resulting binary tree structure will look like this:



As you can see, every node in the transformed structure now has at most two children (a left child and a right child), thus satisfying the definition of a binary tree while preserving the hierarchical relationships of the original general tree.

Question no. 3: What are Red Black Trees? How do they differ from Splay Trees? What are their applications?

Answer:

Red-Black Trees: A Red-Black Tree (RBT) is a sophisticated type of self-balancing binary search tree. Each node within an RBT holds an additional piece of information: its color, which is either red or black. These color attributes are not arbitrary; they are meticulously maintained through a set of rules (properties) during insertion and deletion operations. The

of these properties is to ensure that the tree remains approximately balanced, guaranteeing that fundamental operations like searching, inserting, and deleting nodes can always be performed with a worst-case time complexity of O(log n), where 'n' represents the number of nodes in the tree. This strong guarantee makes RBTs a reliable choice for applications requiring consistent performance.

The five critical properties that define a Red-Black Tree are:

Every node is either red or black.

The root node is always black.

Every leaf (NIL node, which represents empty subtrees and is usually implicit or a sentinel node) is black.

If a node is red, then both its children must be black. This property prevents consecutive red nodes along any path.

Every simple path from a given node to any of its descendant leaf nodes contains the same number of black nodes. This "black-height" property is crucial for maintaining balance.

How do they differ from Splay Trees? Red-Black Trees and Splay Trees are both self-balancing binary search trees, but they achieve balance and optimize performance in fundamentally different ways:

Red-Black Trees maintain a strict balance through explicit color rules and rotations whenever an insertion or deletion violates these rules. This rigid adherence to properties guarantees a worst-case time complexity of O(log n) for individual search, insertion, and deletion operations. However, this

strictness makes their implementation somewhat more complex due to the various cases and rotations that need to be handled. After an access, the structure of a Red-Black Tree only changes to maintain its balance properties, not necessarily to bring the accessed node closer to the root.

Splay Trees, in contrast, use a self-adjusting mechanism called "splaying." When a node is accessed (read, inserted, or deleted), the splay operation performs a series of rotations to move that specific node all the way up to the root of the tree. This process implicitly balances the tree over a sequence of operations, leading to an amortized O(log n) time complexity for search, insert, and delete. This means that while a single operation might, in the worst case, take O(n) time, a sequence of m operations will take O(m log n) total time. Splay trees are generally simpler to implement than Red-Black Trees because there's only one core operation (splay) to worry about. Their structure dynamically changes to bring frequently accessed elements closer to the root, which is highly beneficial for applications exhibiting "locality of reference." Splay trees do not store any extra information (like color bits) per node, making them potentially more memory-efficient than RBTs on a per-node basis.

Applications of Red-Black Trees: Red-Black Trees are highly versatile and are used in numerous critical computer science applications where consistent logarithmic time performance is required.

Associative Containers: They are the backbone of many standard library map or

dictionary implementations (e g , std: : map and std: : set in C++, TreeMap in Java).

Database Indexing: Used extensively in database management systems to create and maintain indexes, allowing for very fast data retrieval.

Operating System Internals: Found in the Linux kernel for managing various data structures, including virtual memory regions, process schedulers, and network packets.

Networking: Can be used in network routers to store and quickly look up routing tables.

Interval Trees: Often serve as the underlying data structure for interval trees, which are used to efficiently query for overlapping intervals (e g , in scheduling or genomic analysis).

Compiler Symbol Tables: Employed in compilers to store and retrieve information about variables, functions, and other symbols during parsing and semantic analysis.

Question no. 4: Write a short note on the recent developments in the area of finding shortest path between two nodes of a Graph. Make necessary assumptions.

Answer:

Short Note on Recent Developments in Shortest Path Algorithms

Finding the shortest path between two nodes in a graph remains a cornerstone problem in computer science, driving advancements across diverse fields like navigation, logistics, network routing, and bioinformatics. While

foundational algorithms such as Dijkstra's (for non-negative weights), Bellman-Ford (for negative weights), and Floyd-Warshall (for all-pairs shortestpaths) provide robust solutions, recent developments have largely focused onenhancing their efficiency, scalability, and adaptability to the massive,dynamic, and complex graphs encountered in real-world applications. One significant area of progress revolves around pre-computation and indexing techniques for static graphs. For scenarios demanding numerous shortest path queries on an unchanging graph (like a road network), methods like Highway Hierarchies (HH) and Contraction Hierarchies (CH) have revolutionized query times. HH identifies a hierarchical structure within the graph, allowing queries to quickly navigate "highways" for long-distance paths. CH, on the other hand, strategically "contracts" nodes and adds shortcut edges, enabling microsecond query times for continent-scale road networks. Another related technique, Transit Node Routing (TNR), leverages key "transit nodes" (e.g., major highway intersections) that most long-distance shortest paths pass through, drastically reducing the search space. Furthermore, Customizable Route Planning (CRP) separates the computationally intensive graph preprocessing from the cost function definition, allowing for rapid changes to routing criteria (e.g., fastest vs. shortest path) without full re-computation.

Another crucial development addresses the need for parallel and distributed computation. With the prevalence of multi-core processors and distributed computing clusters, efforts have been made to parallelize classical

algorithms. This includes GPU-based implementations for algorithms like Bellman-Ford on dense graphs, leveraging the massive parallel processing capabilities of graphics cards. For truly massive graphs that exceed single-machine memory, distributed graph processing frameworks (e.g., Apache Spark's GraphX) enable shortest path calculations across clusters, utilizing techniques like message passing and vertex-centric computation.

The challenge of dynamic graphs and real-time updates has also spurred innovation. Traditional algorithms assume a static graph. However, many real-world graphs, such as transportation networks with live traffic or communication networks with fluctuating loads, are constantly changing. Research in dynamic shortest path algorithms aims to efficiently update existing shortest path information after small changes (e.g., an edge weight change or removal) without recalculating everything from scratch. For applications where absolute real-time response is critical and exact answers are too slow, approximation algorithms provide near-optimal paths within strict time constraints.

Finally, there's an emerging interest in integrating machine learning into shortest path problems. This involves using learning models to predict optimal paths, infer better heuristic functions for algorithms like A*, or even to dynamically adapt algorithm parameters based on graph characteristics or query patterns.

Necessary Assumptions for these developments often include:

Graphs can be extremely large (millions to billions of nodes/edges).

Queries are frequent and demand very low latency (e.g., sub-second or even microsecond responses).

The underlying hardware includes multi-core CPUs, GPUs, and distributed computing infrastructures.

The nature of edge weights can vary: positive, negative, or representing complex cost metrics.

In essence, the field has evolved from merely finding a shortest path to finding the shortest path efficiently and scalably in a highly dynamic and resource-constrained environment, often by combining classic graph theory with advanced data structures, parallel computing paradigms, and even machine learning insights.