# DSA QS BANK - Data Structure Algorithms Question Bank

Data Structures And Algorithms (SRM Institute of Science and Technology)

# DSA

Saturday, August 24, 2024 2:37 PM

Q1> Define linear data structures and give examples? Explain non-linear data structures with examples

Linear Data Structures
Linear data structures are those in which data elements are arranged sequentially or linearly, where each element is connected to its previous and next element. This arrangement allows for easy traversal of the data in a single run. Here are some common examples:

1. Array: A collection of elements stored in contiguous memory locations. Each element can be accessed using its index.
2. Linked List: A collection of nodes where each node contains data and a reference to the next node in the sequence.
3. Stack: A collection of elements that follows the Last-In-First-Out (LIFO) principle. Elements are added and removed from the same end.
4. Queue: A collection of elements that follows the First-In-First-Out (FIFO) principle. Elements are added at one end (rear) and removed from the other end (front).

Non-Linear Data Structures
Non-linear data structures are those in which data elements are not arranged sequentially. Instead, they are arranged in a hierarchical or networked manner, allowing for more complex relationships between elements. Here are some common examples:

1. Tree: A hierarchical structure consisting of nodes, where each node has a value and references to its child nodes. Examples include binary trees, AVL trees, and B-trees.
2. Graph: A collection of nodes (vertices) connected by edges. Graphs can be directed or undirected and are used to represent networks, such as social networks or transportation systems.

Q2> Compare and contrast the different types of non-primitive data structures.

Non-primitive data structures are more complex than primitive data structures and are used to store multiple values or more complex structures of data. They can be broadly categorized into linear and non-linear data structures. Here's a comparison and contrast of the different types:
Linear Data Structures

1. Arrays
    - Structure: A collection of elements stored at contiguous memory locations.
    - Access: Direct access to elements using indices.
    - Usage: Suitable for scenarios where the size of the data set is known and fixed.
    - Example: Storing a list of student grades.
2. Linked Lists
    - Structure: A sequence of elements, where each element points to the next.
    - Access: Sequential access; elements are accessed by traversing from the head.
    - Usage: Useful when the size of the data set can change dynamically.
    - Example: Implementing a queue or stack.
3. Stacks
    - Structure: A collection of elements that follows the Last-In-First-Out (LIFO) principle.
    - Access: Only the top element can be accessed directly.
    - Usage: Useful for scenarios like function call management, undo mechanisms.

    - Example: Browser history management.
4. Queues
    - Structure: A collection of elements that follows the First-In-First-Out (FIFO) principle.
    - Access: Only the front and rear elements can be accessed directly.
    - Usage: Suitable for scheduling tasks, managing requests in a server.
    - Example: Print job management.

Non-Linear Data Structures

1. Trees
    - Structure: A hierarchical structure with a root element and child elements.
    - Access: Elements are accessed through parent-child relationships.
    - Usage: Suitable for representing hierarchical data, such as file systems.
    - Example: Binary search trees for efficient searching and sorting.
2. Graphs
    - Structure: A set of nodes connected by edges.
    - Access: Elements are accessed through connections (edges) between nodes.
    - Usage: Useful for representing networks, such as social networks or transportation systems.
    - Example: Shortest path algorithms in navigation systems.

Comparison

- Memory Usage: Linear data structures like arrays use contiguous memory, which can be more efficient but less flexible. Non-linear structures like trees and graphs use dynamic memory allocation, which is more flexible but can be less efficient.
- Access Time: Arrays provide constant-time access to elements, while linked lists, trees, and graphs may require traversal, leading to longer access times.
- Flexibility: Linked lists, trees, and graphs are more flexible in terms of size and structure compared to arrays and stacks.
- Complexity: Non-linear data structures are generally more complex to implement and manage compared to linear data structures.

Q3> Explain the concept of structures and their valid operations.

In programming, a structure (often referred to as a struct) is a user-defined data type that allows you to group different types of variables under a single name. This is particularly useful for organizing complex data in a way that is easy to manage and understand.

Concept of Structures

A structure is defined using the struct keyword, followed by the structure name and a block of code that specifies its members. Each member can be of any data type, including other structures.

Example in C:

```
struct Person {
  char name[50];
  int age;
  float height;
};
```

In this example, Person is a structure with three members: name, age, and height.

Valid Operations on Structures

1. Declaration and Initialization:
    - Declaration: You can declare variables of the structure type.

struct Person person1;

- Initialization: You can initialize structure members using an initializer list or assignment.
  struct Person person1 = {"Alice", 30, 5.5};
  person1.age = 31; // Using assignment

2. Accessing Members:
   - Use the dot operator (.) to access members of a structure.
     printf("Name: %s\n", person1.name);
     printf("Age: %d\n", person1.age);
3. Pointer to Structure:
   - You can create pointers to structures and access members using the arrow operator (->).
     struct Person *ptr = &person1;
     printf("Height: %.2f\n", ptr->height);
4. Nested Structures:
   - Structures can contain other structures as members.

```
struct Address {
   char city[50];
   int zip;
};

struct Person {
   char name[50];
   int age;
   struct Address address;
};

struct Person person2 = {"Bob", 25, {"New York", 10001}};
printf("City: %s\n", person2.address.city);
```

5. Array of Structures:
   - You can create arrays of structures to store multiple records.

```
struct Person people[3] = {
   {"Alice", 30, 5.5},
   {"Bob", 25, 5.9},
   {"Charlie", 35, 6.0}
};
printf("First person: %s\n", people[0].name);
```

Q4> Is it possible to pass structure as an parameter to a function? Explain with relevant examples

Yes, it is possible to pass a structure as a parameter to a function in C. There are two common ways to do this: by value and by reference.

Passing Structure by Value

When you pass a structure by value, a copy of the entire structure is passed to the function. This means any changes made to the structure within the function do not affect the original structure.

Example:

```
#include <stdio.h>
struct Car {
```

```
    char name[30];
    int price;
};
void printCarInfo(struct Car c) {
    printf("Name: %s\n", c.name);
    printf("Price: %d\n", c.price);
}
int main() {
    struct Car car1 = {"Tata", 1021};
    printCarInfo(car1);
    return 0;
}
```

In this example, the printCarInfo function receives a copy of the car1 structure.

Passing Structure by Reference

When you pass a structure by reference, a pointer to the structure is passed to the function. This allows the function to modify the original structure.

Example:

```
#include <stdio.h>
struct Student {
    char name[50];
    int roll;
    float marks;
};
void display(struct Student *student) {
    printf("Name: %s\n", student->name);
    printf("Roll: %d\n", student->roll);
    printf("Marks: %.2f\n", student->marks);
}
int main() {
    struct Student st1 = {"Aman", 19, 8.5};
    display(&st1);
    return 0;
}
```

In this example, the display function receives a pointer to the st1 structure, allowing it to access and modify the original structure.

Q5> Write pseudo code to perform result processing of a student. The student structure contains the following fields. 1) name ii) Regno iii) Mark1, Mark2, Mark3

```
BEGIN
    // Define the structure for Student
    STRUCT Student
        STRING name
        INTEGER Regno
        INTEGER Mark1
        INTEGER Mark2
        INTEGER Mark3
    END STRUCT
```

```
// Function to calculate total marks
FUNCTION CalculateTotalMarks(Student s)
   RETURN s.Mark1 + s.Mark2 + s.Mark3
END FUNCTION

// Function to calculate average marks
FUNCTION CalculateAverageMarks(Student s)
   INTEGER total = CalculateTotalMarks(s)
   RETURN total / 3
END FUNCTION

// Function to determine grade
FUNCTION DetermineGrade(Student s)
   INTEGER average = CalculateAverageMarks(s)
   IF average >= 90 THEN
      RETURN "A"
   ELSE IF average >= 80 THEN
      RETURN "B"
   ELSE IF average >= 70 THEN
      RETURN "C"
   ELSE IF average >= 60 THEN
      RETURN "D"
   ELSE
      RETURN "F"
   END IF
END FUNCTION

// Main program
BEGIN
   // Create a student instance
   Student student1
   student1.name = "John Doe"
   student1.Regno = 12345
   student1.Mark1 = 85
   student1.Mark2 = 90
   student1.Mark3 = 78

   // Process the student's results
   INTEGER totalMarks = CalculateTotalMarks(student1)
   FLOAT averageMarks = CalculateAverageMarks(student1)
   STRING grade = DetermineGrade(student1)

   // Display the results
   PRINT "Name: " + student1.name
   PRINT "Regno: " + student1.Regno
   PRINT "Total Marks: " + totalMarks
   PRINT "Average Marks: " + averageMarks
   PRINT "Grade: " + grade
END
```

END

Q6> Illustrate the application of self referential structures with an example

Self-referential structures are structures that contain a pointer to another structure of the same type. These structures are fundamental in creating complex data structures like linked lists, trees, and graphs.
Example: Singly Linked List
A common example of a self-referential structure is a node in a singly linked list. Each node contains data and a pointer to the next node in the list.
Structure Definition

```
#include <stdio.h>
#include <stdlib.h>
// Define the structure for a node
struct Node {
   int data;
   struct Node* next;
};
// Function to create a new node
struct Node* createNode(int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   newNode->next = NULL;
   return newNode;
}
// Function to print the linked list
void printList(struct Node* head) {
   struct Node* temp = head;
   while (temp != NULL) {
      printf("%d -> ", temp->data);
      temp = temp->next;
   }
   printf("NULL\n");
}
int main() {
   // Create nodes
   struct Node* head = createNode(1);
   head->next = createNode(2);
   head->next->next = createNode(3);
   head->next->next->next = createNode(4);
// Print the linked list
   printList(head);
return 0;
}
```

Application
Self-referential structures like the one used in this example are essential for implementing dynamic data structures. They allow for efficient memory usage and flexible data management, making them suitable for various applications such as:

- Linked Lists: Efficient insertion and deletion operations.

- Trees: Represent hierarchical data structures.
- Graphs: Model networks and relationships.

Q7> How to use dynamic memory allocation for creating multiple records using structures?

Dynamic memory allocation allows you to allocate memory at runtime, which is particularly useful when the number of records is not known in advance. Here's how you can use dynamic memory allocation to create multiple records using structures in C.
Example: Dynamic Memory Allocation for Student Records
Let's consider a structure for storing student records, including name, Regno, Mark1, Mark2, and Mark3.
Structure Definition

```
#include <stdio.h>
#include <stdlib.h>
// Define the structure for a student
struct Student {
   char name[50];
   int Regno;
   int Mark1;
   int Mark2;
   int Mark3;
};
int main() {
   struct Student *students;
   int numStudents, i;
// Ask the user for the number of students
   printf("Enter the number of students: ");
   scanf("%d", &numStudents);
// Allocate memory for the number of students
   students = (struct Student*) malloc(numStudents * sizeof(struct Student));
// Check if memory allocation was successful
   if (students == NULL) {
      printf("Memory allocation failed!\n");
      return 1;
   }
// Input student details
   for (i = 0; i < numStudents; i++) {
      printf("Enter details for student %d\n", i + 1);
      printf("Name: ");
      scanf("%s", students[i].name);
      printf("Regno: ");
      scanf("%d", &students[i].Regno);
      printf("Mark1: ");
      scanf("%d", &students[i].Mark1);
      printf("Mark2: ");
      scanf("%d", &students[i].Mark2);
      printf("Mark3: ");
      scanf("%d", &students[i].Mark3);
   }
```

```
// Display student details
   printf("\nStudent Details:\n");
   for (i = 0; i < numStudents; i++) {
      printf("Student %d\n", i + 1);
      printf("Name: %s\n", students[i].name);
      printf("Regno: %d\n", students[i].Regno);
      printf("Mark1: %d\n", students[i].Mark1);
      printf("Mark2: %d\n", students[i].Mark2);
      printf("Mark3: %d\n", students[i].Mark3);
   }
// Free the allocated memory
   free(students);
return 0;
}
```

Explanation

1. Structure Definition:
    - The Student structure contains fields for name, Regno, Mark1, Mark2, and Mark3.
2. Dynamic Memory Allocation:
    - The malloc function is used to allocate memory for an array of Student structures based on the number of students entered by the user.
    - The allocated memory is checked to ensure it was successful.
3. Input and Output:
    - A loop is used to input details for each student.
    - Another loop is used to display the details of each student.
4. Memory Deallocation:
    - The free function is used to deallocate the memory once it is no longer needed.

Q8> Define a function to multiply two mxn matrices and return the resultant matrix

Q9> Perform matrix addition using DMA

To perform matrix addition using Dynamic Memory Allocation (DMA) in C, you can follow these steps:

1. Dynamically allocate memory for the matrices.
2. Input the elements of the matrices.
3. Add the corresponding elements of the matrices.
4. Store the result in a new dynamically allocated matrix.
5. Print the result.
6. Free the allocated memory.

Here's a C program that demonstrates this process:

```
#include <stdio.h>
#include <stdlib.h>
void addMatrices(int** A, int** B, int** C, int rows, int cols) {
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         C[i][j] = A[i][j] + B[i][j];
      }
   }
}
```

```
int main() {
   int rows, cols;
   printf("Enter the number of rows: ");
   scanf("%d", &rows);
   printf("Enter the number of columns: ");
   scanf("%d", &cols);
// Dynamically allocate memory for matrices
   int** A = (int**)malloc(rows * sizeof(int*));
   int** B = (int**)malloc(rows * sizeof(int*));
   int** C = (int**)malloc(rows * sizeof(int*));
   for (int i = 0; i < rows; i++) {
      A[i] = (int*)malloc(cols * sizeof(int));
      B[i] = (int*)malloc(cols * sizeof(int));
      C[i] = (int*)malloc(cols * sizeof(int));
   }
// Input elements of matrix A
   printf("Enter elements of matrix A:\n");
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         scanf("%d", &A[i][j]);
      }
   }
// Input elements of matrix B
   printf("Enter elements of matrix B:\n");
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         scanf("%d", &B[i][j]);
      }
   }
// Add matrices A and B
   addMatrices(A, B, C, rows, cols);
// Print the result
   printf("Resultant matrix C:\n");
   for (int i = 0; i < rows; i++) {
      for (int j = 0; j < cols; j++) {
         printf("%d ", C[i][j]);
      }
      printf("\n");
   }
// Free allocated memory
   for (int i = 0; i < rows; i++) {
      free(A[i]);
      free(B[i]);
      free(C[i]);
   }
   free(A);
   free(B);
   free(C);
return 0;
```

}

This program dynamically allocates memory for three matrices (A, B, and C), reads the elements of matrices A and B from the user, adds them, and stores the result in matrix C. Finally, it prints the resultant matrix and frees the allocated memory.

Q10> How mathematical notations are used to measure the complexity of an algorithm?

Mathematical notations are essential tools for analyzing and expressing the complexity of algorithms. These notations help us understand how the performance of an algorithm scales with the size of the input. The most commonly used notations are Big O (O), Big Omega (Ω), and Big Theta (Θ). Let's explore each of these:

1. Big O Notation (O)
   - Purpose: Represents the upper bound of the running time of an algorithm.
   - Usage: Describes the worst-case scenario.
   - Example: If an algorithm has a time complexity of ( O(n^2) ), it means that in the worst case, the running time grows at most proportional to the square of the input size.
2. Big Omega Notation (Ω)
   - Purpose: Represents the lower bound of the running time of an algorithm.
   - Usage: Describes the best-case scenario.
   - Example: If an algorithm has a time complexity of ( \Omega(n) ), it means that in the best case, the running time grows at least proportional to the input size.
3. Big Theta Notation (Θ)
   - Purpose: Represents both the upper and lower bounds of the running time of an algorithm.
   - Usage: Describes the average-case scenario.
   - Example: If an algorithm has a time complexity of ( \Theta(n \log n) ), it means that the running time grows proportionally to ( n \log n ) in both the best and worst cases.

How These Notations Are Used

- Time Complexity: Measures how the running time of an algorithm changes with the size of the input.
- Space Complexity: Measures how the memory usage of an algorithm changes with the size of the input.

Examples

1. Constant Time (O(1)): The running time does not change with the input size.
2. Linear Time (O(n)): The running time grows linearly with the input size.
3. Quadratic Time (O(n^2)): The running time grows quadratically with the input size.

Q 11> Explain different types of mathematical notations used for asymptotic analysis

*There are mainly three asymptotic notations:*

1. *Big-O Notation (O-notation)*
2. *Omega Notation (Ω-notation)*
3. *Theta Notation (Θ-notation)*

1. Theta Notation (Θ-Notation):

*Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.*

*.Theta (Average Case) You add the running times for each possible input combination*

*and take the average in the average case.*

2. Big-O Notation (O-notation):
*Big-O notation represents the upper bound of the running time of an algorithm. Therefore, it gives the worst-case complexity of an algorithm.*

*.It is the most widely used notation for Asymptotic analysis.*

*.It specifies the upper bound of a function.*

*.The maximum time required by an algorithm or the worst-case time complexity.*

*.It returns the highest possible output value(big-O) for a given input.*

*.Big-O(Worst Case) It is defined as the condition that allows an algorithm to complete statement execution in the longest amount of time possible.*

3. Omega Notation (Ω-Notation):
*Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.*

*Q 12*> 2. What is time-space tradeoff? Explain with an example.

The time-space tradeoff is a concept in computer science that refers to the balance between the time complexity and space complexity of an algorithm. Essentially, it means that sometimes you can reduce the time an algorithm takes to run by using more memory, or you can reduce the memory usage by allowing the algorithm to take more time.
Example: Sorting Algorithms
Consider two sorting algorithms: Merge Sort and Insertion Sort.

- Merge Sort:
  - Time Complexity: (O(n \log n))
  - Space Complexity: (O(n))
  - Explanation: Merge Sort is efficient in terms of time but requires additional space proportional to the size of the input array for the temporary arrays used during the merge process.
- Insertion Sort:
  - Time Complexity: (O(n^2))
  - Space Complexity: (O(1))
  - Explanation: Insertion Sort is less efficient in terms of time, especially for large datasets, but it requires very little additional space since it sorts the array in place.

Practical Scenario
Imagine you are working with a large dataset that needs to be sorted. If you have limited memory available, you might choose Insertion Sort despite its higher time complexity because it uses less space. On the other hand, if you have ample memory but need the sorting to be done quickly, Merge Sort would be a better choice due to its lower time complexity.