**✅ SAMPLE ANSWERS FOR 10-MARK QUESTIONS**

**1. Explain different types of linked lists with diagrams.**

**Marks: 10**

**Answer:**

Linked Lists are linear data structures where each element (called a node) contains two parts:

* Data
* Pointer to the next node

There are 3 major types of linked lists:

**1. Singly Linked List**

* Each node points to the next node.
* Last node points to NULL.

struct Node {

int data;

struct Node\* next;

};

**Diagram:**

[10 | \*] -> [20 | \*] -> [30 | NULL]

**Operations:** Insertion, Deletion, Traversal.

**2. Doubly Linked List**

* Each node has two pointers: one to the next and one to the previous node.

struct Node {

int data;

struct Node\* prev;

struct Node\* next;

};

**Diagram:**

NULL <- [10] <-> [20] <-> [30] -> NULL

**Advantage:** Bi-directional traversal.

**3. Circular Linked List**

* The last node points to the first node.
* Can be singly or doubly circular.

**Diagram (Singly Circular):**

[10] -> [20] -> [30] -+

^------------------+

**Applications:** Useful in circular scheduling.

**2. Write and explain the algorithm for merge sort. Give an example.**

**Marks: 10**

**Answer:**

**Merge Sort** is a **divide and conquer** sorting algorithm.

**Steps:**

1. Divide the array into halves until each part contains one element.
2. Merge the sorted halves into a single sorted array.

**Algorithm:**

void mergeSort(int arr[], int l, int r) {

if (l < r) {

int m = (l + r) / 2;

mergeSort(arr, l, m);

mergeSort(arr, m + 1, r);

merge(arr, l, m, r);

}

}

**Merge Function:**

void merge(int arr[], int l, int m, int r) {

int i, j, k;

int n1 = m - l + 1;

int n2 = r - m;

int L[n1], R[n2];

for (i = 0; i < n1; i++) L[i] = arr[l + i];

for (j = 0; j < n2; j++) R[j] = arr[m + 1+ j];

i = j = 0;

k = l;

while (i < n1 && j < n2)

arr[k++] = (L[i] <= R[j]) ? L[i++] : R[j++];

while (i < n1) arr[k++] = L[i++];

while (j < n2) arr[k++] = R[j++];

}

**Example:**  
Sort array [38, 27, 43, 3, 9, 82, 10]

1. Divide:  
   [38, 27, 43, 3] and [9, 82, 10]  
   Then again divided recursively.
2. Merge:  
   Sorted result after merges: [3, 9, 10, 27, 38, 43, 82]

**Time Complexity:**

* Best, Average, Worst: **O(n log n)**

**✅ SAMPLE ANSWERS FOR 5-MARK QUESTIONS**

**1. Differentiate between array and linked list.**

**Marks: 5**

| **Feature** | **Array** | **Linked List** |
| --- | --- | --- |
| Memory | Contiguous | Non-contiguous (dynamic) |
| Size | Fixed (static) | Dynamic (grow/shrink easily) |
| Insertion/Deletion | Expensive (shift elements) | Efficient (adjust pointers) |
| Access time | O(1) - Direct access | O(n) - Sequential |
| Memory use | Wastage possible | Efficient |

**2. What is a queue? List its types.**

**Marks: 5**

**Queue:** A linear data structure that follows **FIFO** (First In First Out) principle.

**Basic Operations:**

* enqueue() – insert element
* dequeue() – remove element

**Types:**

1. **Simple Queue** – FIFO
2. **Circular Queue** – Last position connected to the first
3. **Double-ended Queue (Deque)** – Insertion/deletion from both ends
4. **Priority Queue** – Elements served based on priority

**Example in C:**

#define SIZE 5

int queue[SIZE];

int front = -1, rear = -1;

Great! Let's continue with more detailed answers from the **"Data Structures Using C"** question sets. We'll cover both **10-mark** and **5-mark** questions, with each explanation thorough enough to fetch full marks.

**✅ MORE 10-MARK QUESTIONS (continued)**

**3. Explain the operations on a stack with algorithms and examples.**

**Marks: 10**

**Definition:**  
A **stack** is a linear data structure that follows **LIFO** (Last In First Out) principle.

**Basic Operations:**

1. PUSH – Insert an element.
2. POP – Remove the top element.
3. PEEK – View the top element.
4. isEmpty() – Check if stack is empty.
5. isFull() – Check if stack is full (in static implementation).

**Array Implementation:**

#define SIZE 100

int stack[SIZE];

int top = -1;

// PUSH

void push(int val) {

if (top == SIZE - 1)

printf("Stack Overflow");

else

stack[++top] = val;

}

// POP

int pop() {

if (top == -1) {

printf("Stack Underflow");

return -1;

} else {

return stack[top--];

}

}

**Example:**

Initial Stack: Empty

push(10) → [10]

push(20) → [10, 20]

pop() → 20 (Removed) → [10]

peek() → 10

**Applications:**

* Function call management (Recursion)
* Undo operations in editors
* Expression evaluation (Infix to Postfix)

**4. Explain Binary Search Tree. Write algorithms for insertion and deletion.**

**Marks: 10**

**Binary Search Tree (BST):**

* A binary tree where each node has:
  + Left child < Node
  + Right child > Node

**Structure:**

struct Node {

int data;

struct Node \*left, \*right;

};

**Insertion Algorithm:**

struct Node\* insert(struct Node\* root, int key) {

if (root == NULL) {

struct Node\* newNode = malloc(sizeof(struct Node));

newNode->data = key;

newNode->left = newNode->right = NULL;

return newNode;

}

if (key < root->data)

root->left = insert(root->left, key);

else

root->right = insert(root->right, key);

return root;

}

**Deletion Cases:**

1. Node has no children → delete directly
2. Node has one child → replace node with child
3. Node has two children → replace with inorder successor/predecessor

**Example:**  
Insert 50, 30, 70, 20, 40, 60, 80 into BST.

50

/ \

30 70

/ \ / \

20 40 60 80

Delete 30 (has two children):

* Replace with inorder successor (40)

**✅ MORE 5-MARK QUESTIONS**

**3. Define recursion with example.**

**Marks: 5**

**Recursion:**  
A function that calls itself either directly or indirectly.

**Example – Factorial Function:**

int factorial(int n) {

if (n <= 1) return 1;

else return n \* factorial(n - 1);

}

**Working for factorial(4)**

= 4 \* factorial(3)

= 4 \* 3 \* factorial(2)

= 4 \* 3 \* 2 \* factorial(1)

= 4 \* 3 \* 2 \* 1 = 24

**Applications:**

* Factorials, Fibonacci series
* Tree and Graph traversals
* Backtracking algorithms (N-Queens, Sudoku)

**4. What is circular queue? How does it overcome limitations of linear queue?**

**Marks: 5**

**Circular Queue:**  
A linear data structure where the last position is connected back to the first position, forming a circle.

**Advantage Over Linear Queue:**

* In linear queues, once rear reaches the end, no more insertions happen even if front elements are deleted (wasted space).
* Circular queue reuses freed space.

**Conditions:**

isFull: (rear + 1) % size == front

isEmpty: front == -1

**Diagram:**

Queue: [ \_, \_, \_, 10, 20 ]

Front = 3, Rear = 4 → insert at (4+1)%5 = 0

Perfect! Let's continue with **detailed 10-mark and 5-mark answers** from the remaining important topics of **"Data Structures Using C"**, with enough depth and examples to secure full marks in an exam.

**✅ CONTINUED: 10-MARK QUESTIONS**

**5. Explain tree traversal techniques with examples.**

**Marks: 10**

Tree traversal is the process of visiting all nodes in a tree in a specific order.

There are two main types:

* **Depth-First Traversal (DFT)**
  + Inorder
  + Preorder
  + Postorder
* **Breadth-First Traversal (BFT)**
  + Level Order

**🔹 Inorder Traversal (Left → Root → Right)**

void inorder(struct Node\* root) {

if (root != NULL) {

inorder(root->left);

printf("%d ", root->data);

inorder(root->right);

}

}

**Example Tree:**

1

/ \

2 3

**Inorder Output:** 2 1 3

**🔹 Preorder Traversal (Root → Left → Right)**

void preorder(struct Node\* root) {

if (root != NULL) {

printf("%d ", root->data);

preorder(root->left);

preorder(root->right);

}

}

**Output:** 1 2 3

**🔹 Postorder Traversal (Left → Right → Root)**

void postorder(struct Node\* root) {

if (root != NULL) {

postorder(root->left);

postorder(root->right);

printf("%d ", root->data);

}

}

**Output:** 2 3 1

**🔹 Level Order Traversal (Breadth First)**

Implemented using queue:

void levelOrder(struct Node\* root) {

if (root == NULL) return;

struct Queue\* q = createQueue();

enqueue(q, root);

while (!isEmpty(q)) {

struct Node\* current = dequeue(q);

printf("%d ", current->data);

if (current->left) enqueue(q, current->left);

if (current->right) enqueue(q, current->right);

}

}

**6. Explain file handling in C. Write a program to copy contents of one file to another.**

**Marks: 10**

**File Handling Functions in C:**

* fopen(), fclose(), fread(), fwrite(), fgetc(), fputc(), fprintf(), fscanf()

**Modes in fopen():**

* "r" – read
* "w" – write (overwrite)
* "a" – append
* "r+", "w+", "a+" – read + write combinations

**Program to Copy File:**

#include <stdio.h>

int main() {

FILE \*fp1, \*fp2;

char ch;

fp1 = fopen("source.txt", "r");

fp2 = fopen("dest.txt", "w");

if (fp1 == NULL || fp2 == NULL) {

printf("Error opening file.");

return 1;

}

while ((ch = fgetc(fp1)) != EOF)

fputc(ch, fp2);

printf("File copied successfully.");

fclose(fp1);

fclose(fp2);

return 0;

}

**✅ CONTINUED: 5-MARK QUESTIONS**

**5. Explain stack applications.**

**Marks: 5**

**Applications of Stack:**

1. **Expression Evaluation** – Postfix evaluation using stack.
2. **Expression Conversion** – Infix to Postfix/Prefix.
3. **Function Call Management** – Handles recursion with system stack.
4. **Undo Operation** – In text editors.
5. **Backtracking** – In maze, puzzles, and pathfinding.
6. **Parenthesis Matching** – Syntax checking in compilers.

**6. Differentiate between linear and non-linear data structures.**

**Marks: 5**

| **Feature** | **Linear DS** | **Non-Linear DS** |
| --- | --- | --- |
| Structure | Sequential | Hierarchical/Graph-based |
| Examples | Arrays, Stacks, Queues | Trees, Graphs |
| Traversal | One level at a time | Multiple branches |
| Memory Usage | Simple, contiguous | Can be complex |

**7. What is a graph? Mention its types.**

**Marks: 5**

**Graph:**  
A graph is a collection of **nodes (vertices)** and **edges** connecting pairs of nodes.

**Types:**

1. **Directed Graph (Digraph)** – Edges have direction.
2. **Undirected Graph** – Edges do not have direction.
3. **Weighted Graph** – Edges have weights.
4. **Unweighted Graph** – No weights on edges.
5. **Connected/Disconnected** – Based on reachability.
6. **Cyclic/Acyclic** – Presence or absence of cycles.

**Representation Methods:**

* Adjacency Matrix
* Adjacency List

**8. Write algorithm for linear search.**

**Marks: 5**

**Linear Search:**  
Checks each element one by one.

**Algorithm:**

int linearSearch(int arr[], int n, int key) {

for (int i = 0; i < n; i++)

if (arr[i] == key)

return i; // Found at index i

return -1; // Not found

}

**Time Complexity:** O(n)

Great! Let's proceed with the **next set of detailed answers** from the remaining **Data Structures Using C** questions, ensuring clarity and completeness to fetch full marks.

**✅ CONTINUED: 10-MARK QUESTIONS**

**7. Convert infix expression to postfix expression using stack. Explain with algorithm and example.**

**Marks: 10**

**Infix:** Operators are between operands. Example: A + B  
**Postfix:** Operators come after operands. Example: AB+

**💡 Steps to Convert Infix to Postfix:**

1. Use a stack to hold operators.
2. Output operands directly.
3. Apply **precedence** and **associativity** rules.
4. Use parentheses to guide precedence.

**🔹 Precedence Table:**

| **Operator** | **Precedence** | **Associativity** |
| --- | --- | --- |
| ^ | High | Right to Left |
| \* / % | Medium | Left to Right |
| + - | Low | Left to Right |

**🔹 Algorithm:**

1. Scan infix expression from left to right.
2. If operand, add to postfix.
3. If '(', push to stack.
4. If ')', pop until '('.
5. If operator:
   * Pop from stack to postfix if precedence is higher or equal.
   * Then push current operator.

**🔹 Example:**

**Infix:** (A + B) \* (C - D)

**Postfix Conversion:**

Step 1: ( → push

Step 2: A → output

Step 3: + → push

Step 4: B → output

Step 5: ) → pop → + → output

Step 6: \* → push

Step 7: ( → push

Step 8: C → output

Step 9: - → push

Step 10: D → output

Step 11: ) → pop → - → output

Final Postfix: A B + C D - \*

**8. Explain hashing and collision resolution techniques.**

**Marks: 10**

**Hashing:**  
Technique to store and retrieve data in **constant time O(1)** using a **hash function**.

**🔹 Hash Function:**

Maps keys to indexes in a hash table.

int hash(int key) {

return key % size;

}

**🔹 Collisions:**

When two keys hash to the same index.

**🔹 Collision Resolution Techniques:**

**1. Linear Probing**

* Search next available slot.
* Problem: Clustering

index = (hash + i) % size;

**2. Quadratic Probing**

* Gaps increase quadratically to avoid clustering.

index = (hash + i\*i) % size;

**3. Double Hashing**

* Uses a second hash function.

index = (h1(key) + i \* h2(key)) % size;

**4. Separate Chaining**

* Store multiple elements using linked lists at each index.

struct Node {

int key;

struct Node\* next;

};

**9. Explain quick sort algorithm with example.**

**Marks: 10**

**Quick Sort** is a **divide and conquer** sorting algorithm.

**🔹 Steps:**

1. Choose a pivot.
2. Partition array so that:
   * Elements < pivot go left
   * Elements > pivot go right
3. Recursively apply to left and right parts.

**🔹 Partition Code:**

int partition(int arr[], int low, int high) {

int pivot = arr[high];

int i = low - 1;

for (int j = low; j < high; j++) {

if (arr[j] <= pivot) {

i++;

swap(&arr[i], &arr[j]);

}

}

swap(&arr[i + 1], &arr[high]);

return i + 1;

}

**🔹 Quick Sort Function:**

void quickSort(int arr[], int low, int high) {

if (low < high) {

int pi = partition(arr, low, high);

quickSort(arr, low, pi - 1);

quickSort(arr, pi + 1, high);

}

}

**🔹 Example:**

Array: [10, 80, 30, 90, 40]

1. Pivot = 40 → Partition: [10, 30] 40 [80, 90]
2. Recursively sort left and right

Final sorted: [10, 30, 40, 80, 90]

**✅ CONTINUED: 5-MARK QUESTIONS**

**9. What is binary search? Write its algorithm.**

**Marks: 5**

**Binary Search:**  
Search algorithm for **sorted arrays**, reducing range by half at each step.

**🔹 Algorithm:**

int binarySearch(int arr[], int n, int key) {

int low = 0, high = n - 1, mid;

while (low <= high) {

mid = (low + high) / 2;

if (arr[mid] == key)

return mid;

else if (arr[mid] < key)

low = mid + 1;

else

high = mid - 1;

}

return -1; // Not found

}

**🔹 Time Complexity:**

* Best: O(1)
* Worst: O(log n)

**10. What is priority queue?**

**Marks: 5**

**Priority Queue:**  
An abstract data type where **each element has a priority**. Elements are served based on priority (not just insertion order).

**🔹 Characteristics:**

* High priority elements dequeued before low priority ones.
* Can be implemented using:
  + Arrays
  + Heaps (binary heap is most efficient)

**🔹 Example:**

| **Element** | **Priority** |
| --- | --- |
| Task A | 2 |
| Task B | 1 |
| Task C | 3 |

Output order: Task C, Task A, Task B

Excellent! Let’s continue with the remaining **important 10-mark and 5-mark questions** from **Data Structures Using C**, explained in full depth for maximum marks.

**✅ CONTINUED: 10-MARK QUESTIONS**

**10. Explain insertion sort and selection sort with examples.**

**Marks: 10**

**🔹 Insertion Sort**

**Working:**

* Builds the final sorted array one item at a time.
* Elements are compared with sorted part and inserted at the correct position.

**Algorithm:**

void insertionSort(int arr[], int n) {

int i, key, j;

for (i = 1; i < n; i++) {

key = arr[i];

j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j--;

}

arr[j + 1] = key;

}

}

**Example:**  
Array: [5, 2, 4, 6]

Step-by-step:

Pass 1: [2, 5, 4, 6]

Pass 2: [2, 4, 5, 6]

Pass 3: [2, 4, 5, 6] (Already sorted)

**🔹 Selection Sort**

**Working:**

* Finds the minimum element and places it at the beginning.
* Repeats for remaining elements.

**Algorithm:**

void selectionSort(int arr[], int n) {

int i, j, min;

for (i = 0; i < n - 1; i++) {

min = i;

for (j = i + 1; j < n; j++)

if (arr[j] < arr[min])

min = j;

swap(&arr[min], &arr[i]);

}

}

**Example:**  
Array: [29, 10, 14, 37]

Steps:

Pass 1: [10, 29, 14, 37]

Pass 2: [10, 14, 29, 37]

Pass 3: [10, 14, 29, 37]

**11. Explain operations on doubly linked list.**

**Marks: 10**

A **doubly linked list (DLL)** contains three parts:

* prev pointer (to previous node)
* data
* next pointer (to next node)

struct Node {

int data;

struct Node\* prev;

struct Node\* next;

};

**🔹 Operations:**

**1. Insertion at Beginning:**

void insertFront(struct Node\*\* head, int data) {

struct Node\* newNode = malloc(sizeof(struct Node));

newNode->data = data;

newNode->prev = NULL;

newNode->next = \*head;

if (\*head != NULL) (\*head)->prev = newNode;

\*head = newNode;

}

**2. Insertion at End:**

void insertEnd(struct Node\*\* head, int data) {

struct Node\* newNode = malloc(sizeof(struct Node));

struct Node\* last = \*head;

newNode->data = data;

newNode->next = NULL;

if (\*head == NULL) {

newNode->prev = NULL;

\*head = newNode;

return;

}

while (last->next != NULL) last = last->next;

last->next = newNode;

newNode->prev = last;

}

**3. Deletion:**

* Deletion at beginning, end or by value
* Adjust prev and next pointers

**🔹 Advantages over Singly Linked List:**

* Bi-directional traversal
* Easier deletion without complete traversal

**✅ CONTINUED: 5-MARK QUESTIONS**

**11. What is dynamic memory allocation? List functions used.**

**Marks: 5**

**Dynamic Memory Allocation:**  
Allocates memory at runtime using heap memory. Required when the size of data is unknown during compile time.

**Functions:**

| **Function** | **Use** |
| --- | --- |
| malloc() | Allocates specified number of bytes. Uninitialized. |
| calloc() | Allocates and initializes memory to zero. |
| realloc() | Reallocates memory to new size. |
| free() | Deallocates memory. |

**Example:**

int\* ptr = (int\*) malloc(5 \* sizeof(int));

free(ptr);

**12. Difference between stack and queue.**

**Marks: 5**

| **Feature** | **Stack (LIFO)** | **Queue (FIFO)** |
| --- | --- | --- |
| Insertion | Top | Rear |
| Deletion | Top | Front |
| Order | Last in, First out | First in, First out |
| Functions | push(), pop(), peek() | enqueue(), dequeue(), front() |
| Example Use | Recursion, undo actions | Job scheduling, printing |

**13. What is a circular linked list?**

**Marks: 5**

**Circular Linked List:**  
A variation of linked list where the last node points to the first node, forming a circle.

**Types:**

* Singly Circular
* Doubly Circular

**Benefits:**

* Continuous traversal
* Useful in CPU scheduling (Round Robin)

**Example:**

struct Node {

int data;

struct Node\* next;

};

If node3 → node1: node3->next = node1;

**14. Write advantages of linked list over arrays.**

**Marks: 5**

| **Feature** | **Array** | **Linked List** |
| --- | --- | --- |
| Size | Fixed at compile time | Dynamic at runtime |
| Memory | Wasted if not fully used | Efficient usage |
| Insertion/Deletion | Expensive (shifting) | Easier (pointer updates) |
| Overflow | May occur | Can expand as needed |
| Flexibility | Low | High |