CTSD QUESTIN BANK

One line questions for 1 mark:

1)What is a dangling pointer?

A. A dangling pointer is a pointer that doesn't point to a valid memory location. This usually occurs when an object is deleted or deallocated, but the pointer still points to the memory location of the deallocated memory. This can lead to unexpected behavior in a program.

2) Which preprocessor directive is used to terminate conditional compilation?

A. The preprocessor directive used to terminate conditional compilation in C is `#endif`.

3) What is the difference between local and global variables?

A. Local variables are declared within a specific block of code, such as a function or method, and have limited scope and lifetime, existing only within that block.

On the other hand, global variables are declared outside of any function and can be accessed from any part of the program, persisting throughout its execution.

4) Can you access a structure member without the -&gt; operator using a pointer to the structure?

A. Yes, you can access a structure member without the `->` operator by using the dereference operator `*` and the dot operator `.` like this: `(*ptr).member`.

5) Which preprocessor directive is used to declare macros?

A. The preprocessor directive used to declare macros is `#define`.

6) What is dynamic memory allocation?

A. Dynamic memory allocation is the process of allocating memory at runtime using functions such as `malloc()`, `calloc()`, `realloc()`, and `free()` .

7) List any 4 preprocessing directives?

A. Four common preprocessing directives are:

1. `#define`

2. `#include`

3. `#ifdef`

4. `#ifndef`

8) What are Structures?

A. Structures are user-defined data types in C and C++ that allow grouping of variables of different types under a single name for easy handling.

9) What is the use of the keyword "typedef "?

A. The keyword `typedef` is used to create an alias for existing data types, making code more readable and easier to manage.

10) What are preprocessor directives?

A. Preprocessor directives are instructions that are processed by the compiler's preprocessor before the actual compilation begins. They are used for tasks such as including header files, defining macros, and conditional compilation.

11) What is the use of the enum keyword in C?

A. The `enum` keyword in C is used to define an enumeration, which is a user-defined type consisting of a set of named integer constants. This enhances code readability and maintainability by allowing the use of meaningful names instead of numeric values.

12) What is Structure padding?

A. Structure padding, also known as data structure padding or structure alignment, refers to the insertion of empty bytes between members of a structure in order to ensure that each member is properly aligned in memory. This alignment is often necessary for efficient memory access, particularly on architectures that require data to be aligned on certain byte boundaries for optimal performance

13) What is the use of the bit field in C?.

A.In C, bit fields are used to specify the size (in bits) of individual members within a structure. They allow for more efficient use of memory by packing multiple bit fields into a single storage unit, such as a byte or a word. This is particularly useful when dealing with hardware registers, protocol headers, or other situations where memory efficiency is crucial**.**

14) What is meant by an anonymous structure?

A. An anonymous structure is a structure declaration without a tag name. It's used when you don't need to refer to the structure by name, typically within a larger structure or when defining variables**.**

15) What is meant by nested structures?

A. Nested structures refer to structures that are defined within other structures. This allows for a hierarchical organization of data where one structure contains another as a member.

16) What are self referential structures?

A. Self-referential structures are structures that contain a pointer member that points to the same type of structure. This allows for the creation of complex data structures like linked lists, trees, and graphs.

17) What is the use of #include preprocessor directive?

A, The #include preprocessor directive is used to include the contents of another file (typically header files) in the current source file. It's commonly used to include function prototypes, macro definitions, and declarations.

18) What is the use of .(dot) operator?

A. The dot . operator in C is used to access members of a structure or union variable.

19) What is the use of -&gt;(arrow) operator?

A.The arrow -> operator in C is used to access members of a structure or union variable through a pointer to that structure or union.

20) What is the use of pointer variables in C?

A.Pointer variables in C are used to store memory addresses. They are particularly useful for dynamic memory allocation, accessing elements of arrays and structures indirectly, and for implementing data structures like linked lists and trees.

21) What is Searching and Sorting?

A. Searching and sorting are fundamental operations in computer science and data processing. Searching involves finding a specific item (or items) in a collection of data, while sorting involves arranging the elements of a collection in a specific order (e.g., ascending or descending).

22) List out different types of searching.

A.Different types of searching algorithms include:

- Linear Search
- Binary Search
- Interpolation Search
- Hashing (Hash Table)

23) What is a Linked List?

A.A linked list is a linear data structure consisting of a sequence of elements called nodes, where each node contains both data and a reference (or pointer) to the next node in the sequence. Unlike arrays, linked lists do not have a fixed size, and memory allocation for nodes is dynamic.

24) List out different types of the Linked list

A. Different types of linked lists include:

- Singly Linked List
- Doubly Linked List
- Circular Linked List

<u>**Short answer questions for 3 marks:**</u>

1)Explain the use of typedef keyword and #define preprocessor directive.

A. 1) **Use of \`typedef\` keyword and \`#define\` preprocessor directive**:

- **\`typedef\`**: It's used to create a new name for existing data types. This can improve code readability and maintainability. For example:

```
typedef int Int32; // Creating an alias for int

Int32 number = 10;
```

- **`#define`**: It's used to define constants or macros, which are replaced by their values during preprocessing. For example:

```
#define PI 3.14159

float circle_area = PI * radius * radius;
```

2) How to include user defined header files into your program? Give an example.

A. To include a user-defined header file, you use the `#include` directive followed by the filename enclosed in angle brackets or double quotes. For example:

```
#include "my_header.h"
```

3) What are the situations where a pointer becomes a dangling pointer? Explain with an example.

A. A pointer becomes a dangling pointer when it points to memory that has been deallocated or goes out of scope. For example:

```
int *ptr = (int *)malloc(sizeof(int)); // Allocating memory

free(ptr); // Deallocating memory

*ptr = 10; // Dangling pointer
```

4) Explain the enum with an example.

A. `enum` is used to define a set of named constants. For example:

```
enum Weekday {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};

enum Weekday today = Tuesday;
```

5) Explain about Nested Structures with examples?

A. Nested structures are structures within structures. They allow you to represent complex data structures more intuitively. For example:

```
struct Date {

   int day;

   int month;

   int year;

};

struct Employee {

   int empID;

   char name[50];

   struct Date birthDate; // Nested structure

};

struct Employee emp1 = {101, "John Doe", {25, 5, 1990}};
```

6) Explain about macros with an example.

A. Macros are preprocessor directives that define symbolic constants or small code fragments. They are replaced with their definitions before the actual compilation process begins. For example:

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))

int max_value = MAX(10, 20); // replaced with ((10) > (20) ? (10) : (20))
```

7) Explain about Bit fields with examples.

A. Bit fields are used to specify the size (in bits) of individual members within a structure. They allow for more efficient memory usage by packing multiple bit fields into a single storage unit. For example:

```
struct {
    unsigned int isStudent : 1;
    unsigned int isWorker : 1;
    unsigned int age : 6;
} person;
```

8) What is an enumeration? Give an example for enumeration.

A. An enumeration is a user-defined data type consisting of a set of named integer constants. It allows you to create symbolic names for integral values to improve code readability. For example:

```
enum Color {RED, GREEN, BLUE};

enum Color myColor = RED;
```

9) What are self-referential structures? Write the syntax for the same.

A.Self-referential structures are structures that contain a pointer member pointing to the same type of structure. They are commonly used to implement dynamic data structures like linked lists, trees, and graphs. Here's an example syntax:

```
struct Node {
    int data;
    struct Node *next; // Pointer to the same type of structure
};
```

10) Write a C program to demonstrate Enumeration type?

A.
```
#include <stdio.h>

enum Weekday {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};

int main() {
    enum Weekday today = Wednesday;
    printf("Today is ");
    switch (today) {
        case Sunday:
            printf("Sunday");
            break;
        case Monday:
            printf("Monday");
            break;
        case Tuesday:
            printf("Tuesday");
            break;
        case Wednesday:
            printf("Wednesday");
            break;
        case Thursday:
            printf("Thursday");
            break;
        case Friday:
            printf("Friday");
            break;
        case Saturday:
            printf("Saturday");
            break;
    }
    printf("\n");
    return 0;
}
```

11) List out any 3 differences between malloc and calloc functions in dynamic memory allocation?

A. 1. **Initialization**:

- `malloc` allocates uninitialized memory.
- `calloc` allocates zero-initialized memory.

2. **Arguments**:

- `malloc` takes a single argument, which is the size of the memory block to allocate in bytes.

- `calloc` takes two arguments: the number of elements to allocate and the size of each element in bytes.

3. **Efficiency**:

- `malloc` may be slightly more efficient than `calloc` for large allocations since `calloc` initializes memory to zero, which adds overhead.

- `calloc` may be more efficient for small allocations since it initializes memory in bulk.

12) List out any six of the preprocessor directives available in the C programming language?

A. - `#define`

- `#include`
- `#ifdef`
- `#ifndef`
- `#endif`
- `#pragma`

13) Write a C program to pass structure as an argument to the functions using call by value method?

A.

```
#include <stdio.h>
struct Point {
    int x;
    int y;
};
void displayPoint(struct Point p) {
    printf("Point coordinates: (%d, %d)\n", p.x, p.y);
}
int main() {
    struct Point p1 = {10, 20};
    displayPoint(p1);
    return 0;
}
```

14) Write a C program to demonstrate nested structure?

A.

```
#include <stdio.h>
struct Address {
    char city[50];

    char state[50];

};

struct Employee {

    int empID;

    char name[50];

    struct Address empAddress;

};

int main() {

    struct Employee emp1 = {101, "John Doe", {"New York", "NY"}};

    printf("Employee ID: %d\n", emp1.empID);

    printf("Employee Name: %s\n", emp1.name);

    printf("Employee City: %s\n", emp1.empAddress.city);

    printf("Employee State: %s\n", emp1.empAddress.state);

    return 0;

}
```

15) List out any 3 differences between structure and Unions?

A. 1. **Memory Allocation**:

- In structures, each member has its own memory location.

- In unions, all members share the same memory location.

2. **Memory Size**:

- The size of a structure is the sum of the sizes of its members plus any padding.

- The size of a union is the size of its largest member.

3. **Usage**:

- Structures are used to represent a collection of different data types.

- Unions are used when different data types need to be stored in the same memory location, and only one value needs to be accessed at a time.

16) What are Anonymous structures? Give an example.

A. Anonymous structures are structures without a name. They are typically used when the structure is only needed within another structure or union and doesn't need to be referenced elsewhere. Here's an example:

```
struct Employee {
    int empID;

    char name[50];

    struct {

      int day;

      int month;

      int year;

    } joinDate; // Anonymous structure

  };
```

17) List out any 3 differences between static and dynamic memory allocations?

A. 1. **Allocation Time**:

   - Static memory allocation is done at compile-time.

   - Dynamic memory allocation is done at runtime.

  2. **Size Flexibility**:

   - With static allocation, the size of memory is fixed and cannot be changed during program execution.

   - With dynamic allocation, memory size can be adjusted during runtime using functions like `malloc()` and `realloc()`.

  3. **Memory Management**:

   - Static memory allocation is managed by the compiler.

   - Dynamic memory allocation requires manual memory management, including allocation and deallocation.

18) Write an example program to demonstrate embedded nested structure?

A.
```
#include <stdio.h>

  struct Date {

    int day;

    int month;

    int year;

  };

  struct Employee {

    int empID;

    char name[50];

    struct Date joinDate;
  };
  int main() {
      struct Employee emp1 = {101, "John Doe", {25, 5, 1990}};
      printf("Employee ID: %d\n", emp1.empID);
      printf("Employee Name: %s\n", emp1.name);
      printf("Join Date: %d/%d/%d\n", emp1.joinDate.day, emp1.joinDate.month,
emp1.joinDate.year);
      return 0;
  }
```

19) Write a C program to find the maximum and minimum of two numbers using macros?

A.
```
#include <stdio.h>
  #define MAX(a, b) ((a) > (b) ? (a) : (b))
  #define MIN(a, b) ((a) < (b) ? (a) : (b))
  int main() {
      int num1 = 10, num2 = 20;
      printf("Maximum of %d and %d is %d\n", num1, num2, MAX(num1, num2));
      printf("Minimum of %d and %d is %d\n", num1, num2, MIN(num1, num2));
      return 0;
  }
```

20) Explain the difference between arrays of structures and arrays within structures?

A. - **Arrays of Structures**:

- An array of structures is a collection of multiple instances of a structure type stored in a contiguous block of memory.

- Each element of the array is a complete structure with its own set of members.

- **Arrays within Structures**:

- An array within a structure is a member of the structure that is itself an array.

- It allows grouping of related data items together within a single structure.

21) What is Searching? Explain any two different types of searching with an example.

A. Searching involves finding a specific item within a collection of data. Two different types of searching are:

- **Linear Search**:

- In linear search, each element of the collection is sequentially checked until the desired item is found or the end of the collection is reached.

- Example: Searching for an element in an unsorted array.

- **Binary Search**:

- In binary search, the collection must be sorted.

- It works by repeatedly dividing the search interval in half until the target element is found or the interval is empty.

- Example: Searching for an element in a sorted array.

22) What is sorting? Explain Linear search and Binary search with an example.

A. **Sorting** is the process of arranging elements of a collection in a specific order, typically ascending or descending. It makes it easier to search for elements in the collection efficiently.

Linear search:

- In linear search, each element of the collection is sequentially checked until the desired item is found or the end of the collection is reached.

- Example:

```
int linearSearch(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == target)
            return i; // Return index if found
    }
    return -1; // Return -1 if not found
}
```

Binary search:

- In binary search, the collection must be sorted.

- It works by repeatedly dividing the search interval in half until the target element is found or the interval is empty.

- Example:

```
int binarySearch(int arr[], int low, int high, int target) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target)
            return mid; // Return index if found
        else if (arr[mid] < target)
```

```
            low = mid + 1;
         else
            high = mid - 1;
      }
      return -1; // Return -1 if not found
   }
```

23) Write a difference between Linear search and Binary search.

A.   - **Linear Search**:

   - Suitable for unsorted arrays.

   - Time complexity is O(n), where n is the number of elements in the array.

  - **Binary Search**:

   - Suitable for sorted arrays.

   - Time complexity is O(log n), where n is the number of elements in the array.

24) Write a difference between Merge sort and Quick sort.

A.   - **Merge Sort**:

   - Uses the divide-and-conquer strategy.

   - Time complexity is O(n log n) in all cases.

   - Requires additional space for merging.

  - **Quick Sort**:

   - Uses the divide-and-conquer strategy with partitioning.

   - Average time complexity is O(n log n), but worst-case time complexity is O(n^2).

   - In-place sorting algorithm, hence requires less additional space.

25) Write a difference between Selection sort and Insertion sort.

A. - **Selection Sort**:

   - Iterates over the array, finds the minimum element, and swaps it with the element in the current position.

   - Time complexity is O(n^2) in all cases.

   - Inefficient for large datasets.

  - **Insertion Sort**:

   - Builds the sorted array one element at a time by repeatedly taking the next element and inserting it into the correct position.

- Time complexity is O(n^2) in the worst case but can be efficient for small datasets and nearly sorted arrays.

- Stable sorting algorithm.

26) What is a Linked List? Explain Singly, Doubly and Circular Linked List.

A. A linked list is a linear data structure composed of a sequence of elements called nodes. Each node contains two parts: the data itself and a reference (or pointer) to the next node in the sequence. Linked lists offer dynamic memory allocation and efficient insertion and deletion operations, making them useful for scenarios where the size of the data structure may change frequently.

Singly Linked List:

- In a singly linked list, each node contains data and a single pointer to the next node in the sequence.

- The last node's pointer typically points to NULL, indicating the end of the list.

- Traversal of a singly linked list starts from the head (the first node) and follows the next pointers until reaching NULL.

- Example:

```
 head --> [Data|Next] --> [Data|Next] --> [Data|Next] --> NULL
```

Doubly Linked List:

- In a doubly linked list, each node contains data and two pointers: one to the next node and one to the previous node.

- The first node's previous pointer and the last node's next pointer typically point to NULL.

- Traversal of a doubly linked list can be performed in both forward and backward directions, as each node maintains a reference to its previous node.

- Example:

```
 NULL <--> [Data|Prev|Next] <--> [Data|Prev|Next] <--> [Data|Prev|Next] <--> NULL
```

Circular Linked List:

- In a circular linked list, the last node's next pointer points back to the first node (in a singly linked list) or the first node's previous pointer points to the last node (in a doubly linked list).

- This creates a circular structure, where traversal can start from any node and continue until reaching the starting point again.

- Circular linked lists are useful for applications where continuous looping through the list is required.

- Example:

```
 head --> [Data|Next] --> [Data|Next] --> [Data|Next] --> ... --> [Data|Next] --|
    |_____________________________________________________________|
```

__Questions for 4 marks:__

1)What are Macros? Explain macros with an example program?

A.1) **Macros**:

Macros in C are preprocessor directives that define symbolic constants or small code fragments. They are typically used for defining constants, inline functions, or conditional compilation.

Example program:

```
#include <stdio.h>

#define PI 3.14159

#define SQUARE(x) ((x) * (x))

int main() {

    printf("Value of PI: %f\n", PI);

    printf("Square of 5: %d\n", SQUARE(5));

    return 0;

}
```

2) What are structure pointers? .Write a C program that uses structure pointer to refer structure members(attributes) using the arrow (-&gt;) operator?

A. Structure pointers are pointers that point to structures. They are used to access structure members using the arrow (`->`) operator.

Example program:

```
#include <stdio.h>

struct Point {

    int x;

    int y;

};

int main() {

    struct Point p1 = {10, 20};

    struct Point *ptr = &p1;


    printf("Coordinates of p1: (%d, %d)\n", ptr->x, ptr->y);

    return 0;

}
```

3) Write a C program that uses “array of structures” to store 3(three) “BOOK” records based on

“Book_Pages, Book_Price and Book_Name” and to display all the records of 3(three) Books.

A.
```
#include <stdio.h>
struct Book {
    int pages;
    float price;
    char name[50];
};
int main() {
    struct Book books[3];
    // Input data
    for (int i = 0; i < 3; i++) {
        printf("Enter details for Book %d:\n", i + 1);
        printf("Pages: ");
        scanf("%d", &books[i].pages);
        printf("Price: ");
        scanf("%f", &books[i].price);
        printf("Name: ");
        scanf("%s", books[i].name);
    }
    // Displaying data
    printf("\nBook Records:\n");
    for (int i = 0; i < 3; i++) {
        printf("Book %d:\n", i + 1);
        printf("Pages: %d\n", books[i].pages);
        printf("Price: %.2f\n", books[i].price);
        printf("Name: %s\n", books[i].name);
    }
    return 0;
}
```

4) Explain an Array of pointers with syntax and examples.

A. An array of pointers is an array where each element is a pointer. It's typically used to store multiple pointers to different objects.

Syntax:

data_type *array_name[size];

Example:

int *ptr_array[5];

5) Explain about malloc(), calloc() and realloc() with examples? Differentiate between malloc()and calloc().

A. - `malloc()`: Allocates memory of specified size in bytes and returns a pointer to the allocated memory.

- `calloc()`: Allocates memory for an array of elements, initializes them to zero, and returns a pointer to the allocated memory.

- `realloc()`: Changes the size of previously allocated memory. It can either expand or shrink the memory block.

Example:

int *arr1 = (int *)malloc(5 * sizeof(int)); // Allocating memory for 5 integers

int *arr2 = (int *)calloc(5, sizeof(int)); // Allocating memory for 5 integers, initialized to zero

// Reallocating memory for 10 integers

arr1 = (int *)realloc(arr1, 10 * sizeof(int));

Difference between `malloc()` and `calloc()`:

- `malloc()` does not initialize the allocated memory, while `calloc()` initializes the allocated memory to zero.

6) What do you mean by conditional compilation in preprocessing directives? Explain with examples.

A.**Conditional compilation** is a feature provided by preprocessors in many programming languages (such as C and C++) that allows code to be included or excluded from the compilation process based on certain conditions. This is typically done using preprocessing directives.

The most common directives used for conditional compilation are:

- `#ifdef` (if defined)

- `#ifndef` (if not defined)

- `#if` (if)

- `#else`

- `#elif` (else if)

- `#endif`

Example:

```
#include <stdio.h>

// Define a macro

#define DEBUG

int main() {

   int x = 5, y = 10;

#ifdef DEBUG

   printf("Debug: x = %d, y = %d\n", x, y);

#endif

   int result = x + y;

   printf("Result = %d\n", result);

   return 0;

}
```

7) Explain the concept of pointers to structures and self referential structure with examples.

A.**Pointers to Structures:**

Pointers to structures allow you to create dynamic structures and access their members using pointers.

Example:

```
#include <stdio.h>

#include <stdlib.h>

struct Point {

   int x;

   int y;

};

int main() {

   struct Point *p1 = (struct Point *)malloc(sizeof(struct Point));

   p1->x = 10;

   p1->y = 20;

   printf("Point coordinates: (%d, %d)\n", p1->x, p1->y);

   free(p1);

   return 0;
}
```

Self-Referential Structures:

A self-referential structure is a structure that contains a pointer to an instance of the same structure type. This is often used in linked lists and trees.

Example:

```
#include <stdio.h>

#include <stdlib.h>

struct Node {

   int data;

   struct Node *next;

};

int main() {

   struct Node *head = (struct Node *)malloc(sizeof(struct Node));

   head->data = 1;

   head->next = NULL;

   printf("Node data: %d\n", head->data);

   free(head);

   return 0;

}
```

8) Create one structure named Employee with empID, empName, and empSalary. Create another structure named Address with city, pincode and street inside the Employee structure. Take user input and print the result.

A.#include <stdio.h>

```
struct Address {

   char city[50];

   int pincode;

   char street[100];

};

struct Employee {

   int empID;

   char empName[50];

   float empSalary;
```

```
    struct Address empAddress;
};
int main() {
    struct Employee emp;
    // Taking user input
    printf("Enter Employee ID: ");
    scanf("%d", &emp.empID);
    printf("Enter Employee Name: ");
    scanf("%s", emp.empName);
    printf("Enter Employee Salary: ");
    scanf("%f", &emp.empSalary);
    printf("Enter Employee City: ");
    scanf("%s", emp.empAddress.city);
    printf("Enter Employee Pincode: ");
    scanf("%d", &emp.empAddress.pincode);
    printf("Enter Employee Street: ");
    scanf("%s", emp.empAddress.street);
    // Printing the result
    printf("\nEmployee Details:\n");
    printf("ID: %d\n", emp.empID);
    printf("Name: %s\n", emp.empName);
    printf("Salary: %.2f\n", emp.empSalary);
    printf("City: %s\n", emp.empAddress.city);
    printf("Pincode: %d\n", emp.empAddress.pincode);
    printf("Street: %s\n", emp.empAddress.street);
    return 0;
}
```

9) Create a Student structure with student name, student roll number and date of birth. Take input for 5students and print the result.

A.**Student Structure:**

```
#include <stdio.h>
```

```
struct DateOfBirth {
    int day;
    int month;
    int year;
};
struct Student {
    char name[50];
    int rollNumber;
    struct DateOfBirth dob;
};
int main() {
    struct Student students[5];
    // Taking input for 5 students
    for (int i = 0; i < 5; i++) {
        printf("Enter details for student %d:\n", i + 1);
        printf("Name: ");
        scanf("%s", students[i].name);
        printf("Roll Number: ");
        scanf("%d", &students[i].rollNumber);
        printf("Date of Birth (dd mm yyyy): ");
        scanf("%d %d %d", &students[i].dob.day, &students[i].dob.month, &students[i].dob.year);
    }
    // Printing the result
    printf("\nStudent Details:\n");
    for (int i = 0; i < 5; i++) {
        printf("Student %d:\n", i + 1);
        printf("Name: %s\n", students[i].name);
        printf("Roll Number: %d\n", students[i].rollNumber);
        printf("Date of Birth: %02d-%02d-%04d\n", students[i].dob.day, students[i].dob.month,
students[i].dob.year);
    }
    return 0;
}
```

10) What is structure padding? Explain with an example?

A.**Structure padding** is the process by which the compiler adds extra bytes between members of a structure to align the data in memory according to the architecture's alignment requirements. This can result in unused memory space but improves access speed by aligning data to word boundaries.

Example:

```
#include <stdio.h>
struct Example {
    char a;
    int b;
    char c;
};
int main() {
    struct Example ex;
    printf("Size of structure Example: %lu\n", sizeof(ex));
    return 0;
}
```

11) Write an example program to return a structure from functions?

A.#include <stdio.h>

```
// Define a structure
struct Point {
    int x;
    int y;
};
// Function that returns a structure
struct Point createPoint(int x, int y) {
    struct Point p;
    p.x = x;
    p.y = y;
    return p;
}
```

```
int main() {
   struct Point p1 = createPoint(10, 20);
   printf("Point coordinates: (%d, %d)\n", p1.x, p1.y);
   return 0;
}
```

12) Write an example program to pass a structure as an argument to the functions?

A.#include <stdio.h>

```
// Define a structure
struct Rectangle {
   int length;
   int width;
};
// Function to calculate the area of a rectangle
int calculateArea(struct Rectangle r) {
   return r.length * r.width;
}
int main() {
   struct Rectangle rect;
   rect.length = 10;
   rect.width = 5;
   int area = calculateArea(rect);
   printf("Area of the rectangle: %d\n", area);
   return 0;
}
```

13) Perform Selection sort and Insertion sort in given arrays:

I. a[5] = {28,25,99,1}

II. x[7] = {39,25,43,2,1,28}

A.**Selection Sort:**

```
#include <stdio.h>
// Function to perform Selection Sort
void selectionSort(int arr[], int n) {
    int i, j, min_idx;
    for (i = 0; i < n-1; i++) {
        min_idx = i;
        for (j = i+1; j < n; j++) {
            if (arr[j] < arr[min_idx]) {
                min_idx = j;
            }
        }
        int temp = arr[min_idx];
        arr[min_idx] = arr[i];
        arr[i] = temp;
    }
}
int main() {
    int a[5] = {28, 25, 99, 1};
    int n = sizeof(a)/sizeof(a[0]);
    selectionSort(a, n);
    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");

    return 0;
}
```

**Insertion Sort:*

```
#include <stdio.h>
// Function to perform Insertion Sort
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 = j - 1;
        }
        arr[j + 1] = key;
    }
}
int main() {
    int x[7] = {39, 25, 43, 2, 1, 28};
    int n = sizeof(x)/sizeof(x[0]);
    insertionSort(x, n);
    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", x[i]);
    }
    printf("\n");
    return 0;
}
```

14) Perform Quick sort and Merge sort in given arrays:

I. a[5] = {28,25,99,1}

II. x[7] = {39,25,43,2,1,28}

A.**Quick Sort:**

```
#include <stdio.h>
// Function to partition the array
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return (i + 1);
}
// Function to perform Quick Sort
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);
    }
}
int main() {
    int a[5] = {28, 25, 99, 1};
    int n = sizeof(a)/sizeof(a[0]);
    quickSort(a, 0, n-1);
    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
    return 0;
}
```

Merge Sort:

```
#include <stdio.h>
// Function to merge two subarrays
void merge(int arr[], int l, int m, int r) {
  int n1 = m - l + 1;
  int n2 = r - m;
  int L[n1], R[n2];
  for (int i = 0; i < n1; i++) {
    L[i] = arr[l + i];
  }
  for (int j = 0; j < n2; j++) {
    R[j] = arr[m + 1 + j];
  }
  int i = 0, j = 0, k = l;
  while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
      arr[k] = L[i];
      i++;
    } else {
      arr[k] = R[j];
      j++;
    }
    k++;
  }
  while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
  }
  while (j < n2) {
    arr[k] = R[j];
    j++;
```

```
        k++;
    }
}
// Function to perform Merge Sort
void mergeSort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}
int main() {
    int x[7] = {39, 25, 43, 2, 1, 28};
    int n = sizeof(x)/sizeof(x[0]);
    mergeSort(x, 0, n-1);
    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", x[i]);
    }
    printf("\n");
    return 0;
}
```

15) Perform Bubble sort in given arrays:

I. a[5] = {28,25,99,1}

II. x[7] = {39,25,43,2,1,28}

A.**Bubble Sort:**

```
#include <stdio.h>
// Function to perform Bubble Sort
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
```

```c
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
int main() {
    int a[5] = {28, 25, 99, 1};
    int n = sizeof(a)/sizeof(a[0]);
    bubbleSort(a, n);
    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", a[i]);
    }
    printf("\n");
    return 0;
}
#include <stdio.h>
// Function to perform Bubble Sort
void bubbleSort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}
int main() {
    int x[7] = {39, 25, 43, 2, 1, 28};
    int n = sizeof(x)/sizeof(x[0]);
    bubbleSort(x, n);
    printf("Sorted array: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", x[i]);
    }
    printf("\n");
    return 0;
}
```

16) Convert given array into singly linked list and perform below mentioned operations:

I. a[5] = {28,25,99,1}

a. Insert new element 88 after 99

b. Remove 25 from the list

c. Update 1 as 17

A. #include <stdio.h>

```
#include <stdlib.h>
// Define a node for the doubly linked list
struct Node {
    int data;
    struct Node* next;
    struct Node* prev;
};
// 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;
    newNode->prev = NULL;
    return newNode;
}
// Function to print the doubly linked list
void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d <-> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
// Function to insert a node after a given node
void insertAfter(struct Node* prevNode, int data) {
    if (prevNode == NULL) {
        printf("The given previous node cannot be NULL.\n");
        return;
    }
    struct Node* newNode = createNode(data);
    newNode->next = prevNode->next;
    newNode->prev = prevNode;
    if (prevNode->next != NULL) {
        prevNode->next->prev = newNode;
    }
    prevNode->next = newNode;
}
// Function to remove a node with given data
void removeNode(struct Node** head, int data) {
    struct Node* temp = *head;
    while (temp != NULL && temp->data != data) {
        temp = temp->next;
    }
    if (temp == NULL) return;
    if (temp->prev != NULL) {
        temp->prev->next = temp->next;
    } else {
        *head = temp->next;
    }
    if (temp->next != NULL) {
        temp->next->prev = temp->prev;
    }
    free(temp);
}
// Function to update a node with given data
void updateNode(struct Node* head, int oldData, int newData) {
    struct Node* temp = head;
    while (temp != NULL) {
        if (temp->data == oldData) {
            temp->data = newData;
            return;
        }
        temp = temp->next;
    }
}
int main() {
    int arr[] = {39, 25, 43, 2, 1, 28};
    int n = sizeof(arr) / sizeof(arr[0]);
    struct Node* head = createNode(arr[0]);
    struct Node* temp = head;
    for (int i = 1; i < n; i++) {
        struct Node* newNode = createNode(arr[i]);
        temp->next = newNode;
        newNode->prev = temp;
        temp = newNode;
    }
    printf("Original list: ");
    printList(head);
    // Insert new element 88 after 1
    temp = head;
    while (temp != NULL && temp->data != 1) {
        temp = temp->next;
    }
    if (temp != NULL) {
        insertAfter(temp, 88);
    }
    printf("After inserting 88 after 1: ");
    printList(head);
    // Remove 25 from the list
    removeNode(&head, 25);
    printf("After removing 25: ");
    printList(head);
    // Update 43 as 17
    updateNode(head, 43, 17);
    printf("After updating 43 to 17: ");
    printList(head);
    return 0;
}
```

17) Convert given array into doubly and circular linked list and perform below mentioned operations:

I. x[7] = {39,25,43,2,1,28}

a. Insert new element 88 after 1

b. Remove 25 from the list

c. Update 43 as 17

A. #include <stdio.h>

```c
#include <stdlib.h>
// Define a node for the singly linked list
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");
}
// Function to insert a node after a given node
void insertAfter(struct Node* prevNode, int data) {
   if (prevNode == NULL) {
      printf("The given previous node cannot be NULL.\n");
      return;
   }
   struct Node* newNode = createNode(data);
   newNode->next = prevNode->next;
```

```
    prevNode->next = newNode;
}
// Function to remove a node with given data
void removeNode(struct Node** head, int data) {
    struct Node* temp = *head, *prev = NULL;
    if (temp != NULL && temp->data == data) {
        *head = temp->next;
        free(temp);
        return;
    }
    while (temp != NULL && temp->data != data) {
        prev = temp;
        temp = temp->next;
    }
    if (temp == NULL) return;
    prev->next = temp->next;
    free(temp);
}
// Function to update a node with given data
void updateNode(struct Node* head, int oldData, int newData) {
    struct Node* temp = head;
    while (temp != NULL) {
        if (temp->data == oldData) {
            temp->data = newData;
            return;
        }
        temp = temp->next;
    }
}
int main() {
    int arr[] = {28, 25, 99, 1};
    int n = sizeof(arr) / sizeof(arr[0]);
    struct Node* head = createNode(arr[0]);
    struct Node* temp = head;
    for (int i = 1; i < n; i++) {
        temp->next = createNode(arr[i]);
        temp = temp->next;
    }
    printf("Original list: ");
    printList(head);
    // Insert new element 88 after 99
    temp = head;
    while (temp != NULL && temp->data != 99) {
        temp = temp->next;
    }
    if (temp != NULL) {
        insertAfter(temp, 88);
    }
    printf("After inserting 88 after 99: ");
    printList(head);
    // Remove 25 from the list
    removeNode(&head, 25);
    printf("After removing 25: ");
    printList(head);
    // Update 1 as 17
    updateNode(head, 1, 17);
    printf("After updating 1 to 17: ");
    printList(head);\
    return 0;
}
```

Questions for 5 marks:

1) Write a C program to find a maximum and minimum number in an array using dynamic memory allocation.

A.
```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int n, i;
    int *arr;
    int max, min;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    arr = (int *)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Memory allocation failed\n");
        return 1;
    }
    printf("Enter the elements: \n");
    for (i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }
    max = min = arr[0];
    for (i = 1; i < n; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
        if (arr[i] < min) {
            min = arr[i];
        }
    }
    printf("Maximum number is %d\n", max);
    printf("Minimum number is %d\n", min);
    free(arr);
    return 0;
}
```

`2) Explain the declaration and initialization and accessing of structures with syntax and examples.

A.**Declaration and Initialization:**

```
#include <stdio.h>
// Structure declaration
struct Student {
    int id;
    char name[50];
    float marks;
};
// Structure initialization
struct Student s1 = {1, "John Doe", 85.5};
int main() {
    // Accessing structure members
    printf("ID: %d\n", s1.id);
    printf("Name: %s\n", s1.name);
    printf("Marks: %.2f\n", s1.marks);
    return 0;
}
```

Explanation:

- **Declaration**: The `struct Student` is declared with three members: `id`, `name`, and `marks`.
- **Initialization**: The structure `s1` is initialized with values `1`, `"John Doe"`, and `85.5`.
- **Accessing**: The members of the structure are accessed using the dot operator (e.g., `s1.id`).

3) Explain an array of structures and how arrays are represented within a structure.

A.Array of Structures:****

```
#include <stdio.h>
struct Student {
    int id;
    char name[50];
    float marks;
};
int main() {
    struct Student students[3] = {
        {1, "John Doe", 85.5},
        {2, "Jane Smith", 90.0},
        {3, "Emily Davis", 78.5}
    };
    for (int i = 0; i < 3; i++) {
        printf("ID: %d, Name: %s, Marks: %.2f\n", students[i].id, students[i].name, students[i].marks);
    }
    return 0;
}
```

Arrays within a Structure:

```
#include <stdio.h>
struct Course {
    char name[30];
    int scores[5];
};
int main() {
    struct Course course1 = {"Math", {85, 90, 78, 92, 88}};
    printf("Course Name: %s\n", course1.name);
    printf("Scores: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", course1.scores[i]);
    }
    printf("\n");
    return 0;
}
```

Explanation:

- **Array of Structures**: The `students` array holds three `Student` structures. Each element in the array is accessed and printed.

- **Arrays within a Structure: The `Course` structure contains an array `scores`. The scores are printed by accessing the `course1.scores` array.**

4) How to include user defined header files into your program? Give an example.

A.**Example:**

```
// myheader.h

#ifndef MYHEADER_H

#define MYHEADER_H

void greet();

#endif // MYHEADER_H

// myheader.c

#include <stdio.h>

#include "myheader.h"

void greet() {

   printf("Hello, World!\n");

}

// main.c

#include "myheader.h"

int main() {

   greet();

   return 0;

}
```

Explanation:

- `myheader.h`: The header file contains the declaration of the `greet` function.

- `myheader.c`: The implementation of the `greet` function.

- `main.c`: The main program includes the user-defined header file and calls the `greet` function.

5) Create a 2D array dynamically. Take user input and print only even numbers.

A.#include <stdio.h>

```
#include <stdlib.h>

int main() {

   int rows, cols;

   int **arr;
```

```
printf("Enter the number of rows: ");
scanf("%d", &rows);
printf("Enter the number of columns: ");
scanf("%d", &cols);
// Allocate memory for rows
arr = (int **)malloc(rows * sizeof(int *));
for (int i = 0; i < rows; i++) {
  // Allocate memory for columns in each row
  arr[i] = (int *)malloc(cols * sizeof(int));
}
// Input elements
printf("Enter the elements of the array:\n");
for (int i = 0; i < rows; i++) {
  for (int j = 0; j < cols; j++) {
    scanf("%d", &arr[i][j]);
  }
}
// Print even numbers
printf("Even numbers in the array:\n");
for (int i = 0; i < rows; i++) {
  for (int j = 0; j < cols; j++) {
    if (arr[i][j] % 2 == 0) {
      printf("%d ", arr[i][j]);
    }
  }
}
printf("\n");
// Free allocated memory
for (int i = 0; i < rows; i++) {
  free(arr[i]);
}
```

```
    free(arr);
    return 0;
}
```

6) Create a program to perform arithmetic operations using macros.

A.
```
#include <stdio.h>
#define ADD(x, y) ((x) + (y))
#define SUBTRACT(x, y) ((x) - (y))
#define MULTIPLY(x, y) ((x) * (y))
#define DIVIDE(x, y) ((y) != 0 ? (x) / (y) : 0)  // Check for divide by zero
int main() {
    int a, b;
    printf("Enter two numbers: ");
    scanf("%d %d", &a, &b);
    printf("Addition: %d + %d = %d\n", a, b, ADD(a, b));
    printf("Subtraction: %d - %d = %d\n", a, b, SUBTRACT(a, b));
    printf("Multiplication: %d * %d = %d\n", a, b, MULTIPLY(a, b));
    if (b != 0) {
        printf("Division: %d / %d = %d\n", a, b, DIVIDE(a, b));
    } else {
        printf("Division by zero is not allowed.\n");
    }
    return 0;
}
```

7) Explain with the help of diagrams how structures are different than unions in terms of representation of memory?

8) Write a C program to store and print the following strings as an array of pointers to characters and explain how the strings are represented in the memory with the help of a diagram?

Strings: “Parul”, “University”, “Vadodara”

A
```
#include <stdio.h>
int main() {
    // Array of pointers to characters
    char *strings[] = {"Parul", "University", "Vadodara"};
```

```
   int i;
   // Printing the strings
   for (i = 0; i < 3; i++) {
      printf("%s\n", strings[i]);
   }
   return 0;
}
 strings:  [0]    [1]        [2]
           |      |          |
           v      v          v
```

- `strings[0]` points to the 'P' of "Parul".

- `strings[1]` points to the 'U' of "University".

- `strings[2]` points to the 'V' of "Vadodara".

9) What is the need of Dynamic memory allocation in C?

A. Dynamic memory allocation in C is essential for managing memory efficiently, especially when the size of the data structure is not known at compile time and can change during runtime[12]. It allows programs to get memory space at runtime which is very useful for data structures whose size cannot be determined beforehand, like linked lists and dynamic arrays[15].

One of the library functions used for dynamic memory allocation is `malloc()`. It allocates a single large block of memory with the specified size and returns a pointer of type `void` which can be cast into a pointer of any form. Here's an example:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
   int *ptr;
   int n, i;
   // Get the number of elements from the user
   printf("Enter number of elements: ");
   scanf("%d", &n);
   // Dynamically allocate memory using malloc()
   ptr = (int*)malloc(n * sizeof(int));
   // Check if the memory has been successfully allocated by malloc or not
   if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    } else {
        printf("Memory successfully allocated using malloc.\n");
        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;  // Store consecutive integers
        }
        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d ", ptr[i]);
        }
    }
    return 0;
}
```

10) What are Structures and Unions? Explain with an example?

A.Structures and Unions are both user-defined data types in C. Structures (`struct`) allow grouping of variables of different types under a single name, whereas Unions (`union`) allow storing different data types in the same memory location but only one at a time[1].

Here's an example of a structure and a union:

```
#include <stdio.h>
// Define a structure
struct StructExample {
    int integer;
    float decimal;
    char name[20];
};
// Define a union
union UnionExample {
    int integer;
    float decimal;
    char name[20];
};
int main() {
    // Initialize a structure
    struct StructExample s = {18, 38.0, "Copilot"};
    // Initialize a union
    union UnionExample u;
    u.decimal = 38.0; // Only the last assigned member 'decimal' has the valid data
    // Print structure and union values
    printf("Structure integer: %d\n", s.integer);
    printf("Union decimal: %.2f\n", u.decimal);
    return 0;
}
```

11) Explain the need for structures in C language with an example?

A.The need for structures in C arises when there is a requirement to group related data items together. For instance, if you want to store information about a book, you might want to keep the title, author, and ISBN number together. Structures allow you to create a data type that encapsulates all these items[6].

Here's an example:

```
#include <stdio.h>
// Define a structure for a book
struct Book {
    char title[50];
    char author[50];
    int isbn;
};
int main() {
    // Declare a book and initialize it
    struct Book book1 = {"The C Programming Language", "Kernighan and Ritchie", 123456789};
    // Access and print book information
    printf("Book Title: %s\n", book1.title);
    printf("Book Author: %s\n", book1.author);
    printf("Book ISBN: %d\n", book1.isbn);

    return 0;

}
```

12) What are pointers in structures? Explain with an example program how the structure members are accessed using structure pointers?

A.Pointers in structures are used to access the members of a structure through their memory address. This is particularly useful when passing structures to functions or managing dynamic memory allocation. Here's an example program that demonstrates accessing structure members using structure pointers:

```
#include <stdio.h>

// Define a structure

struct Student {

    int roll_no;

    char name[30];

    float marks;

};

int main() {

    struct Student s1 = {27, "John Doe", 72.5};

    struct Student *ptr = &s1; // Pointer to the structure

    // Accessing structure members using pointer

    printf("Roll No: %d\n", ptr->roll_no);

    printf("Name: %s\n", ptr->name);

    printf("Marks: %.2f\n", ptr->marks);

    return 0;

}
```

13) What is Searching? List out all the different types of Searching and explain any three with an example.

A.Searching is the process of finding an element within a data structure. There are various types of searching algorithms, including:

- **Linear Search**

- **Binary Search**

- **Jump Search**

- **Interpolation Search**

- **Exponential Search**

- **Fibonacci Search**

- **Ternary Search**

1. **Linear Search**: It is a simple search algorithm that checks every element in the list until the desired element is found or the list ends.

```
int linearSearch(int arr[], int n, int x) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == x) return i;
    }
    return -1;
}
```

2. **Binary Search**: This algorithm is more efficient than linear search and works on sorted arrays by repeatedly dividing the search interval in half.

```
int binarySearch(int arr[], int l, int r, int x) {
    while (l <= r) {
        int m = l + (r - l) / 2;
        if (arr[m] == x) return m;
        if (arr[m] < x) l = m + 1;
        else r = m - 1;
    }
    return -1;
}
```

3. **Jump Search**: This algorithm creates a block and tries to find the element in that block. If the element is not found, it jumps to the next block.

```
int jumpSearch(int arr[], int x, int n) {
    int step = sqrt(n);
    int prev = 0;
    while (arr[min(step, n) - 1] < x) {
        prev = step;
        step += sqrt(n);
        if (prev >= n) return -1;
    }
```

```
   while (arr[prev] < x) {
      prev++;
      if (prev == min(step, n)) return -1;
   }
   if (arr[prev] == x) return prev;
   return -1;
}
```

Explain Insertion and Selection sort with an example.

A. - Selection sort divides the array into a sorted and an unsorted portion. It repeatedly selects the minimum element from the unsorted portion and swaps it with the first element of the unsorted portion.

- Example:

```
    void selectionSort(int arr[], int n) {
       for (int i = 0; i < n-1; i++) {
          int min_idx = i;
          for (int j = i+1; j < n; j++) {
             if (arr[j] < arr[min_idx]) {
                min_idx = j;
             }
          }
          int temp = arr[min_idx];
          arr[min_idx] = arr[i];
          arr[i] = temp;
       }
    }
```

3. *Insertion Sort*:

- Insertion sort builds the sorted array one element at a time by repeatedly taking the next element from the unsorted portion and inserting it into its correct position in the sorted portion.

- Example:

```
    void insertionSort(int arr[], int n) {
       for (int i = 1; i < n; i++) {
          int key = arr[i];
```

```
            int j = i - 1;

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

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

                j = j - 1;

            }

            arr[j + 1] = key;

        }

    }
```

15) Explain Merge and Quick sort with an example.

A.. *Merge Sort*:

- Merge sort divides the array into smaller subarrays, recursively sorts each subarray, and then merges the sorted subarrays to produce the final sorted array.

- Example:

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

        // Merge two sorted subarrays arr[l..m] and arr[m+1..r]

    }

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

        if (l < r) {

            int m = l + (r - l) / 2;

            mergeSort(arr, l, m);

            mergeSort(arr, m + 1, r);

            merge(arr, l, m, r);

        }

    }
```

5. *Quick Sort*:

- Quick sort selects a pivot element, partitions the array into two subarrays (elements smaller than the pivot and elements larger than the pivot), and recursively sorts each subarray.

- Example:

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

        // Partition the array around a pivot

    }
```

```
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);
    }
}
```

16) What is a Linked List? Explain all the different types of Linked List with an example.

A. Linked List

A linked list is a linear data structure where elements, called nodes, are stored in non-contiguous memory locations. Each node contains two parts:

1. **Data:** The value stored in the node.

2. **Pointer:** A reference to the next node in the sequence.

Unlike arrays, linked lists allow for efficient insertion and deletion of elements as they do not require shifting elements.

Types of Linked Lists

1. **Singly Linked List**

2. **Doubly Linked List**

3. **Circular Linked List**

1. Singly Linked List

In a singly linked list, each node contains a data part and a pointer to the next node in the sequence. The last node points to `NULL`, indicating the end of the list.

Structure:

```
struct Node {
    int data;
    struct Node* next;
};
```

Example:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
void printList(struct Node* n) {
    while (n != NULL) {
        printf("%d -> ", n->data);
        n = n->next;
    }
    printf("NULL\n");
}
int main() {
    struct Node* head = NULL;
    struct Node* second = NULL;
    struct Node* third = NULL;
    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));
    head->data = 10;
    head->next = second;
    second->data = 20;
    second->next = third;
    third->data = 30;
    third->next = NULL;
    printList(head);
    return 0;
}
```

2. Doubly Linked List

In a doubly linked list, each node contains three parts:

1. Data

2. Pointer to the next node

3. Pointer to the previous node

Structure:

```
struct Node {
   int data;
   struct Node* next;
   struct Node* prev;
};
```

Example:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
   int data;
   struct Node* next;
   struct Node* prev;
};
void printList(struct Node* node) {
   struct Node* last;
   printf("Traversal in forward direction: ");
   while (node != NULL) {
      printf("%d -> ", node->data);
      last = node;
      node = node->next;
   }
   printf("NULL\n");
   printf("Traversal in reverse direction: ");
   while (last != NULL) {
      printf("%d -> ", last->data);
      last = last->prev;
   }
   printf("NULL\n");
}
int main() {
   struct Node* head = NULL;
```

```
    struct Node* second = NULL;
    struct Node* third = NULL;
    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));
    head->data = 10;
    head->next = second;
    head->prev = NULL;
    second->data = 20;
    second->next = third;
    second->prev = head;
    third->data = 30;
    third->next = NULL;
    third->prev = second;
    printList(head);
    return 0;
}
```

3. Circular Linked List

In a circular linked list, the last node points back to the first node, forming a circle.

Structure:

```c
struct Node {
    int data;
    struct Node* next;
};
```

Example:

```
#include <stdio.h>
#include <stdlib.h>
struct Node {
    int data;
    struct Node* next;
};
void printList(struct Node* head) {
    struct Node* temp = head;
    if (head != NULL) {
        do {
            printf("%d -> ", temp->data);
            temp = temp->next;
        } while (temp != head);
    }
    printf("(head)\n");
}
int main() {
    struct Node* head = NULL;
    struct Node* second = NULL;
    struct Node* third = NULL;
    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));
    third = (struct Node*)malloc(sizeof(struct Node));
    head->data = 10;
    head->next = second;
    second->data = 20;
    second->next = third;
    third->data = 30;
    third->next = head;
    printList(head);
    return 0;
}
```

17) Convert the array into doubly and circular linked list and perform below mentioned operations:

X[10] = {29,21,28,23,27,25}

a. Insert new elements 20 and 24.

b. Update the element 27 as 22

c. Remove 25 from the list.

A. #include <stdio.h>

```
#include <stdlib.h>
// Define the structure for a doubly linked list node
struct Node {
   int data;
   struct Node* prev;
   struct Node* next;
};
// Function to insert a new node at the end of the list
struct Node* insertAtEnd(struct Node* start, int data) {
   struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = data;
   if (start == NULL) { // If the list is empty, create the first node
      newNode->next = newNode->prev = newNode;
      start = newNode;
   } else {
      struct Node* last = start->prev;
      newNode->next = start;
      start->prev = newNode;
      newNode->prev = last;
      last->next = newNode;
   }
   return start;
}
// Function to update a node in the list
struct Node* updateNode(struct Node* start, int oldVal, int newVal) {
   struct Node* current = start;
   if (start == NULL) return start;
   do {
      if (current->data == oldVal) {
            current->data = newVal;
            break;
        }
        current = current->next;
    } while (current != start);
    return start;
}
// Function to delete a node from the list
struct Node* deleteNode(struct Node* start, int data) {
    if (start == NULL) return NULL;
    struct Node *current = start, *prev_1 = NULL, *temp = NULL;
    // Check if the node to be deleted is the start node
    if (current->data == data) {
        prev_1 = start->prev; // Last node
        temp = start;
        prev_1->next = start->next;
        start = start->next;
        start->prev = prev_1;
        free(temp);
        return start;
    }
    // Check other nodes
    prev_1 = start;
    current = start->next;
    while (current->data != data && current != start) {
        prev_1 = current;
        current = current->next;
    }
    if (current->data == data) {
        prev_1->next = current->next;
        current->next->prev = prev_1;
        free(current);
    }
    return start;
}
// Function to display the list
void displayList(struct Node* start) {
    struct Node* temp;
    if (start == NULL) {
        printf("List is empty.\n");
        return;
    }
    temp = start;
    printf("The elements in the list are: ");
    do {
        printf("%d ", temp->data);
        temp = temp->next;
    } while (temp != start);
    printf("\n");
}
int main() {
    int arr[] = {29, 21, 28, 23, 27, 25};
    int n = sizeof(arr) / sizeof(arr[0]);
    struct Node* start = NULL;
    // Create the circular doubly linked list
    for (int i = 0; i < n; i++) {
        start = insertAtEnd(start, arr[i]);
    }
    // Insert new elements 20 and 24
    start = insertAtEnd(start, 20);
    start = insertAtEnd(start, 24);
    // Update the element 27 to 22
    start = updateNode(start, 27, 22);

    // Remove 25 from the list

    start = deleteNode(start, 25);

    // Display the final list

    displayList(start);

    return 0;

}
```